

Communication Infrastructure Modeling of Many-Core Architectures

Vom Fachbereich Elektrotechnik und Informationstechnik
der Technischen Universität Darmstadt
zur Erlangung des akademischen Grades eines
Doktor–Ingenieurs (Dr.-Ing.)
genehmigte Dissertation

M.Sc.
Leandro Heleno Möller
geboren am 7. Februar 1981
in Porto Alegre, Brasilien

Referent:	Prof. Dr. Dr. h. c. mult. Manfred Glesner
Korreferent:	Prof. Dr.-Ing. Hans Eveking Prof. Dr.-Ing. Leandro Soares Indrusiak
Tag der Einreichung:	01.11.2011
Tag der mündlichen Prüfung:	20.12.2011

D17
Darmstadt, 2012

Acknowledgements

Several times I catch myself thinking on all choices that I have made in my life to arrive here. During this path, I remember that on many occasions I wanted to travel back in time and change things to the way I wanted. Today, I feel glad that I did not have this power, because no other sequence of choices could have brought me to such a fantastic outcome. Not only I'm very happy with the outcome, but also with the path itself, where I have met many nice people on the way.

Prof. Manfred Glesner has surpassed all my expectations. I came to Germany without knowing him personally and luckily I got an amazing advisor that has received me with open heart in his institute. Thank you for all the oportunities that you have made available for me and the time you have spent with me. Dr. Leandro Soares Indrusiak is an essential person in this path. He not only shares the same name as I and comes from the same state in Brazil, but he has also shared several hours of Java programming. He is also the leader of the penguin team, consisted of me, Sanna Määttä and Luciano Ost. We have worked together for about an year at the Microelectronics System institute (MES) and we have had remarkable conversations at lunch time in the university restaurant. Sanna, the only girl that I have ever met that enjoys a temperature of -15°C, was a good company during the PhD. Ost is one of those dificult guys to work with. He is always jumping on his chair and singing while we are trying to work! The worse thing is that I had to daily stand him from 2002 to 2008, since I was already sharing room with him before the PhD! I also thank my colleagues Heiko Hinkelmann, Christopher Spies, Sebastian Pankalla and Hans-Peter Keil for all the help with specialities of the german language and german law. The PhD time at the MES was great! I thank all my colleagues for the nice time together! Talking about a great time, my good friend and neighbor Rafael Huff was always near for a beer or a eurotrip! Prof. Fernando Moraes was the one that remained with the difficult job to teach me all about FPGAs, VHDL, computer architecture and writing papers. We have been working together since 2000 and I'm very grateful for our good friendship and more than 35 papers that we have published together. Prof. Ney Calazans was always present with his deep knowledge and adding the final touch that makes our work special. I would like to thank also the professors Hans Eveking, Silvia Santini and Anja Klein for the constructive comments and the evaluation of my work.

My mother, father and brother deserve the highest degree of gratitude for all the support and love in these last 30 years. For me the beginning of the PhD came together with a new life in a new continent. And as a romantic movie that turns into reality, I met my wife on day 1! Laura, thank you for bringing me so much happiness in this amazing journey! On day 1454 our son Lorenzo was born! Thank you for making our days colorful with your wonderful smile!

Abstract

Many-core architectures are becoming a standard design alternative for embedded systems. The force that is driving to this direction is the contradiction that improving the battery lifetime of a chip requires a reduction of the power consumption, but improving the performance of a chip by increasing the clock frequency increases the power consumption. As there is no solution for this problem, the alternative is to introduce several cores to a chip and make them work in parallel.

However, going from single-core to many-core architectures is not straightforward and this is the main concern of this thesis. It requires both new programming methodologies for using multiple cores in parallel and an efficient communication infrastructure to interconnect these cores. A monitoring system connected to the communication infrastructure is also recommended to provide feedback to dynamic task mapping and task migration algorithms.

This thesis contemplates the following issues related to many-core architectures: creation of a many-core architecture model with emphasis on the communication infrastructure, modeling of applications over the many-core architecture model, support for a heterogeneous many-core architecture model, implementation of task mapping and migration algorithms, implementation of monitoring systems, and two different designs of a dual-layer Network-on-Chip that provides Quality-of-Service.

Kurzfassung

So genannte "Many-Core"-Architekturen stellen für eingebettete Systeme den neuesten Stand der Technik dar und werden schon bald Standardlösungen darstellen. Die Entwicklung dieser Architekturen wurde von der Erkenntnis getrieben, dass es einen unauflösbaren Widerspruch zwischen dem Wunsch nach geringerem Energieverbrauch und dem nach der Steigerung der Rechenleistungen von Ein-Kern-Prozessoren durch Erhöhung der Taktfrequenz gibt. Die Einführung von "Multi-Core"- mit einigen wenigen und schließlich "Many-Core"-Architekturen mit zahlreichen parallel arbeitenden Rechenkernen ist der einzige Weg, die Rechenleistung von Prozessoren weiter zu steigern.

Der Übergang von Ein-Kern- auf Mehr-Kern-Architekturen ist keineswegs trivial. Die vorliegende Arbeit befasst sich daher mit einigen der bei diesem Übergang auftretenden Herausforderungen: Eine neue Programmiermethodik ist zur Ausnutzung mehrerer parallel arbeitender Rechenkerns ebenso erforderlich wie eine effiziente Infrastruktur zur Kommunikation zwischen denselben. Die Messung bestimmter Zustandsvariablen der Kommunikationsinfrastruktur hilft dabei, zur Laufzeit einen Lastausgleich zwischen den einzelnen Kernen durchzuführen.

Die vorliegende Arbeit behandelt die Modellierung von Mehr-Kern-Architekturen, wobei der Kommunikationsinfrastruktur besondere Aufmerksamkeit zuteil wird und heterogene Architekturen ausdrücklich unterstützt werden, die Modellierung von Anwendungen für Mehr-Kern-Architekturen, Algorithmen für die Zuordnung neuer Prozesse zu Rechenkernen ("task mapping") und für die Migration laufender Prozesse ("task migration"), die Implementierung eines Kontrollsystems für das Kommunikationsnetzwerk sowie zwei verschiedene mögliche Architekturen eines On-Chip-Netzwerks mit zwei Ebenen mit Dienstgütegarantie.

Table of Contents

1	Introduction	1
1.1	Goals and Contributions.....	2
1.2	Background.....	3
1.3	Thesis Outline.....	4
2	NoC Modeling and Monitoring	7
2.1	Modeling	7
2.1.1	Actor-Oriented	8
2.1.2	Ptolemy II.....	9
2.2	Communication Infrastructure Modeling.....	10
2.2.1	Related Works	10
2.2.2	NoC Infrastructures	11
2.2.3	HERMES	13
2.2.4	RENATO.....	16
2.2.5	JOSELITO.....	19
2.2.6	BOÇA.....	24
2.2.7	Comparison of Platform Models	25
2.3	Monitoring and Debugging for NoC models	27
2.3.1	PointToPointScope	29
2.3.2	EndToEndScope.....	30
2.3.3	InputScope.....	31
2.3.4	OutputScope.....	31
2.3.5	HotspotScope	31
2.3.6	PowerScope	31
2.3.7	BufferScope.....	32
2.4	Monitoring and Debugging for RLT NoCs.....	32
2.4.1	BufferScope.....	35
2.4.2	InputScope.....	35
2.4.3	OutputScope.....	35
2.4.4	EndToEndScope.....	35
2.4.5	PointToPointScope	36
2.4.6	HotSpotScope	36

2.4.7	PowerScope	36
2.5	Summary	36
3	Application and MPSoC Modeling	39
3.1	Application Modeling	39
3.1.1	Synthetic Traffic	40
3.1.2	Instruction Set Simulator	40
3.1.3	UML Sequence Diagrams	46
3.2	Heterogeneous MPSoC	48
3.2.1	Type System	49
3.2.2	Task Parameters	50
3.2.3	Multi-Task Manager	52
3.2.4	UsageScope	53
3.3	Summary	53
4	Task Mapping and Migration	55
4.1	Related Works	56
4.2	Initial Task Mapping	57
4.2.1	First Free (FF)	57
4.2.2	Cluster (CL)	58
4.3	Dynamic Task Mapping	59
4.3.1	First Free (FF)	60
4.3.2	Nearest Neighbor (NN)	60
4.3.3	Minimum Maximum Channel Load (MMC)	61
4.3.4	Minimum Average Channel Load (MAC)	64
4.3.5	Path Load (PL)	65
4.3.6	Best Neighbor (BN)	67
4.3.7	Minimum Data Exchange (MDE)	67
4.3.8	Cost Based (CB)	69
4.4	Comparison of the Dynamic Mapping Algorithms	70
4.5	Task Migration	73
4.5.1	Communication Task Graph Migration	74
4.5.2	Runtime Communication Task Graph Migration	74
4.5.3	HotSpotScope Migration	75
4.5.4	Efficiency Migration	75

4.5.5	Surplus Migration.....	75
4.5.6	Results	76
4.6	Summary	76
5	Reconfigurable Dual-Layer NoC	79
5.1	Dynamic Partial Reconfiguration	79
5.1.1	Introduction to Partial Reconfiguration	80
5.1.2	The Early Access Partial Reconfiguration flow	80
5.2	Proposed Architecture	84
5.3	Communication Protocol.....	86
5.4	Expected advantages and disadvantages.....	87
5.5	Implementation.....	87
5.5.1	Choice of FPGA and software	87
5.5.2	Hierarchy of the system.....	87
5.5.3	Routers	89
5.6	System-on-Chip.....	91
5.6.1	IP cores	91
5.6.2	Dual-layer NoC.....	100
5.7	Early Access Partial Reconfiguration design flow	103
5.7.1	Synthesis	105
5.7.2	Floorplanning.....	105
5.7.3	PR Implementation.....	107
5.8	SoC simulation and test	109
5.8.1	SoC simulation.....	109
5.9	Timing analysis of the SoC.....	114
5.9.1	Timing analysis in simulation.....	114
5.9.2	Timing analysis of the real system.....	115
5.10	Area analysis	123
5.11	Summary.....	125
6	Non-Reconfigurable Dual-Layer NoC	127
6.1	Proposed Architecture	127
6.2	Communication Protocol.....	127
6.3	Expected advantages and disadvantages.....	128
6.4	Implementation.....	129

6.4.1	Choice of FPGA and software	129
6.4.2	Hierarchy of the system.....	129
6.4.3	Control router.....	129
6.4.4	Data router.....	134
6.5	System-on-Chip.....	134
6.5.1	Test core	134
6.5.2	Dual-layer NoC.....	135
6.5.3	Network interface	137
6.6	Simulation.....	143
6.6.1	Data communication established successfully	143
6.6.2	Data communication not established - target busy	145
6.6.3	Data communication not established - router busy.....	148
6.7	Timing analysis of the SoC	151
6.8	Area analysis	153
6.9	Summary	155
7	Conclusions and Future Works	157

Abbreviations

ALU	Arithmetic Logic Unit
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ASIC	Application Specific Integrated Circuits
ASIP	Application Specific Instruction Set Processor
BM	Bus Macro
BN	Best Neighbor
CAD	Computer-Aided Design
CB	Cost Based
CL	Cluster
CLB	Complex Logic Block
CPU	Central Processing Unit
CTG	Communication Task Graph
DCM	Digital Clock Manager
DE	Discrete Event
DMA	Direct Memory Access
DSP	Digital Signal Processor
EAPR	Early Access Partial Reconfiguration
FF	First Free
FIFO	First In First Out
Flit	Flow Control Digit
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GALS	Globally-Asynchronous Locally-Synchronous
GPP	General Purpose Processor

HDL	Hardware Description Language
IC	Integrated Circuit
ICAP	Internal Configuration Access Port
IDE	Interactive Development Environment
IP	Intellectual Property
ISS	Instruction Set Simulator
JVM	Java Virtual Machine
LSB	Least Significant Bit
LUT	Look-Up Table
MAC	Minimum Average Channel Load
MDE	Minimum Data Exchange
MIPS	Microprocessor without Interlocked Pipeline Stages
MMC	Minimum Maximum Channel Load
MoC	Model of Computation
MPSoC	Multiprocessor System-on-Chip
MSB	Most Significant Bit
NI	Network Interface
NN	Nearest Neighbor
NoC	Network-on-Chip
PAT	Payload Abstraction Technique
PE	Processing Element
PL	Path Load
PR	Partial Reconfiguration
PRM	Partial Reconfiguration Module
PRR	Partial Reconfiguration Region
SAF	Shift and Forward
SoC	System-on-Chip
QoS	Quality of Service
RAM	Random Access Memory
RMI	Remote Method Invocation
ROM	Read Only Memory
RTL	Register Transfer Level

TCL	Tool Command Language
TLM	Transaction Level Modeling
UCF	User Constraints File
UML	Unified Modeling Language
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits
VM	Virtual Machine
XML	Extensible Markup Language

1 Introduction

Many-core architectures have started to become a standard in the computer industry with the release of the Intel Pentium D processor in 2005 [1]. Since then, processor manufacturers have focused on many-core architectures to raise the processing power, favoring a larger number of cores working in parallel instead of increasing the clock frequency of a single-core, which is no longer achievable due to excessive power consumption and heat dissipation issues regarding the current technology.

Going from single-core to many-core architectures opens the door to many possibilities but also pose unique **programming challenges**. While executing several small applications in parallel have a significant improve in performance on many-core architectures, a unique complex application needs a careful development to use wisely this processing power. It is not a case of simply writing the application code with multiple threads, but each thread has to be really executing in the same time as the other threads, instead of paused in a *wait* directive.

The synchronicity of threads running over multiple cores also poses a **communication challenge**. While communication infrastructures based on bus have been sufficient for systems composed by a dozen of cores, the increasing number of cores on a chip and the data transfer associated to them will demand a more complex on-chip interconnection. For this purpose Networks-on-Chip (NoCs) have arisen as a scalable solution to future increase on the number of cores. Clearly NoCs with optimized schemes will be employed to provide Quality of Service (QoS) among the communicating cores.

Not only a NoC with QoS is required for an efficient communication between cores, but also communicating threads should be located near to each other to minimize the allocation of NoC resources, posing a severe **mapping challenge** in between the hardware and software universes of many-core architectures. The problem is minimized if it is known before execution which threads communicate with each others, then static or dynamic mapping algorithms can be used with a good success rate. As this is usually not the case, task migration algorithms can be employed at runtime to reduce the most congested parts of the many-core architecture.

Most of these task mapping and migration algorithms require some kind of feedback from the platform, informing them which parts are heavily congested and should be prioritized. This brings a **monitoring challenge**, because NoCs contain links spread throughout the chip, demanding a complex probing of different components of the

platform. Besides, due to the distributed nature of many-core architectures, all monitoring data should not be sent to a centralized point of the system to be analyzed, endangering the creation of a bottleneck.

1.1 Goals and Contributions

All the challenges pointed previously (i.e. programming, communication, mapping and monitoring) are issues contemplated on this thesis and play an important role on many core architectures. For these issues to be properly evaluated, the first goal of this thesis is to create an MPSoC model that is not only flexible enough to accept a huge range of parameters to describe both the application and the platform, but also accurate enough to provide valuable information about the platform when modeling applications with specific constraints. As creating an MPSoC model needs to consider so many aspects that would be alone a PhD thesis, this task was shared by the author of this thesis, Dr. Luciano Copello Ost, Dr. Sanna Määttä and Dr. Leandro Soares Indrusiak.

The second goal of this thesis is to expand the MPSoC model to a heterogeneous MPSoC model, and add dynamic task mapping and task migration capabilities to it. The heterogeneous MPSoC will allow us to model MPSoCs composed by different types of IP cores, where usually most of them are for general use (e.g. processor) and a few others are for specific use (e.g. video decoder, crypto core). The dynamic task mapping and task migration features will improve the performance of the system by assuring that the communicating tasks are not located far from each other or using too many resources of the network for no reason.

After analyzing applications modeled on the created MPSoC and even though improvements are achieved with dynamic mapping and task migration heuristics, the standard packet switching NoC finally limits the performance of the system by not providing QoS. For this reason, the third goal of this thesis is to create a NoC capable of providing guaranteed throughput for some of the communicating IP core connected to it. This is demonstrated by what was called a dual-layer NoC, which is optimized to be a low area overhead communication infrastructure.

During the pursuit of these goals, the following contributions can be highlighted:

- Implementation of an actor-oriented NoC model called RENATO
- Implementation of an actor-oriented NoC model called JOSELITO, which is faster to simulate than RENATO
- Implementation NoC monitors, allowing an improved observability of traffic flows passing through the network
- Support and interface of the previously existing MARS Instruction Set Simulator (ISS) as a processing element connected the RENATO NoC
- Interface of UML sequence diagrams as a source of data traffic to the RENATO NoC

- Implementation of an heterogeneous MPSoC composed by different types of IP cores connected to the RENATO NoC
- Implementation of dynamic task mapping algorithms for the heterogeneous MPSoC
- Implementation of task migration algorithms for the heterogeneous MPSoC
- Implementation of a reconfigurable dual-layer NoC supporting QoS
- Implementation of a non-reconfigurable dual-layer NoC supporting QoS

1.2 Background

Before diving into the advanced topics of this thesis, this Section presents the most important definitions and concepts that allow associating the language of this document to the literature related to MPSoCs.

IP core: A complex digital system pre-designed, pre-verified and prototyped in hardware at least once. IP cores are used as components of an application, being their reusability in different applications an important characteristic on the design of digital systems.

Communication infrastructure: It refers to the resource(s) used to interconnect different IP cores. Examples of communication infrastructures are point-to-point, bus and Network-on-Chip.

Point-to-point communication infrastructure: It is a kind of communication infrastructure that connects IP cores using wire only, establishing a direct connection between them.

Bus communication infrastructure: It is a kind of communication infrastructure that uses the same set of wires that are shared between IP cores.

Network-on-Chip (NoC): It is a kind of communication infrastructure based on routers interconnected by links. These routers are responsible to relay information from an IP core source to the correct IP core target.

System-on-Chip (SoC): A system composed of different IP cores and implemented on a single integrated circuit, providing fast communication between IP cores.

Multiprocessor System-on-Chip (MPSoC): It is a SoC composed by several processors and interconnected by a communication infrastructure.

Homogeneous MPSoC: It is an MPSoC composed by several processors of a unique type.

Heterogeneous MPSoC: It is an MPSoC composed by different types of IP cores, where at least one type of IP core is a processor.

Task: It is a specific operation to be performed. Usually tasks communicate with other tasks to achieve certain functionality. In software a task is frequently called thread, in

hardware a task is frequently called process.

Task mapping: Process of defining to which IP core of an MPSoC a task should be executed. A good or a bad mapping may influence the performance of the MPSoC on different ways, e.g. total execution time, power consumption and generation of hot spots in the system.

Static task mapping: The task mapping is performed at design time and all tasks are already mapped on the system as soon as it starts.

Dynamic task mapping: The task mapping is performed at runtime and each task is mapped as soon as required, possibly taking into consideration the available resources of the system as source information for the task mapping decision.

Task migration: Process of redefining to which IP core of an MPSoC a task should be transferred. This is important because the dynamics of applications may start to change after some time executing in the system, and that mapping done earlier is no longer optimal due to new tasks mapped to the system or new workloads generated by the tasks.

Application: It is a set of tasks that are grouped together to perform a specific functionality.

1.3 Thesis Outline

Fig. 1.1 illustrates a limited graphical view of how this thesis is organized in Chapters. Numbers on the figure refers to Chapters of this thesis. Application is represented by a graph, where nodes represent tasks and edges represent communication between tasks. The platform is represented by IP cores interconnected by NoC routers. The rest of this Section provides a brief overview of which topics are approached.

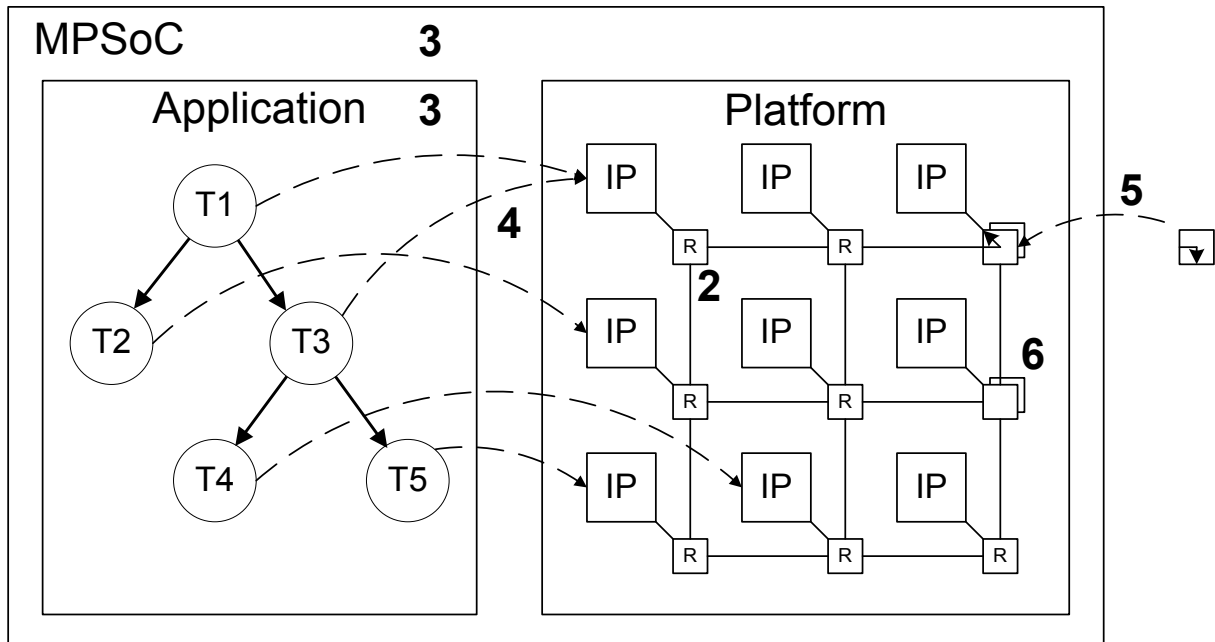


Fig. 1.1: A limited graphical view of how this thesis is organized in Chapters.

Chapter 2: This Chapter starts with an overview about modeling tools and techniques. Ptolemy II is introduced as a framework that uses actor-orientation to build and simulate models following different model of computations. Next, other NoC models created and based on the HERMES NoC are explained. After that, monitoring and debugging techniques created for both RTL and abstract NoC models are presented.

Chapter 3: Different ways used to create applications and to execute them jointly with the created NoCs are presented on this Chapter. The implementation of a heterogeneous MPSoC model composed by different types of IP cores is explained.

Chapter 4: Several dynamic task mapping and task migration algorithms are presented. These algorithms are evaluated over the heterogeneous MPSoC presented on the previous Chapter.

Chapter 5: In order to improve the Quality-of-Service of the NoC, a dual-layer NoC targeting partially and dynamically reconfigurable devices is introduced.

Chapter 6: The dual-layer NoC presented on the previous Chapter is extended to work with non-reconfigurable devices.

Chapter 7: Final conclusions and future works are presented.

2 NoC Modeling and Monitoring^{*}

The main components of electronic systems are no longer easily distinguishable as in the past. Everything is already so fused that the main components are interlaced with other components that optimize them. The reason behind is efficiency, miniaturization and increased number of functionalities. The only way to understand and optimize them further is to have computer models that simulate the real machine, thus allowing us humans to modify these models, evaluate these models, and expect that the improvements made on the models can be reproduced when applied to the real machines.

This thesis is no different. The goal is to first implement an MPSoC model, for better understanding all the complexities of each part while considering it as whole, for later optimizing the most promising parts of it.

As implementing a complete prototype of an optimized many-core architecture is impossible for a unique person or even a small group of people, this Chapter of this thesis presents different NoC models for many-core architectures. Different NoC models were created to allow trading simulation time with accuracy of behavior and timing results. After that, monitoring systems that help evaluating and debugging the NoC are presented. From now on “many-core architecture” will be referred as MPSoC, which is a short for Multi-Processor System-on-Chip.

2.1 Modeling

Register-Transfer Level (RTL) is a known level of abstraction to describe digital circuits, which usually uses a hardware description language (HDL) to represent the circuit. Circuits described at the RTL can be automatically synthesize and translated by using a sequence of Computer-Aided Design (CAD) tools to come up with a solid prototype of the circuit on devices like Application Specific Integrated Circuits (ASICs) and Field Programmable Gate Arrays (FPGAs).

Above RTL, the Transaction-Level Modeling (TLM) stands out for modeling the functionality of the circuit with a well-defined way of abstracting the communication among IP cores and separating it from computation blocks. TLM also uses function calls, making it ideal for hardware/software co-verification at early stages of development as

^{*} Major parts of this Chapter were presented at SBCCI [101], ISVLSI [102], IDT [106] and BEC [110].

a virtual platform.

Section 2.1.1 presents another abstraction level for modeling circuits which is based on actors. Actor-orientation preserves the main qualities of TLM by separating computation from communication and using function calls, but it allows also multiple models of computation. Section 2.1.2 presents in more detail a mature and open-source framework for creating actor-oriented models.

2.1.1 Actor-Orientation

The main components behind actor-orientation are the actors, which execute their own programming and have a well-defined interface. This interface is represented by ports and is used to communicate with other actors through channels, as illustrated on Fig. 2.1. A composition of interconnected actors is called model. A model also has a well-defined interface represented by ports, allowing the creation of hierarchical models. A hierarchical model can be implemented by adding a composite actor to the model, thus allowing the creation of a “sub-model”. In contrast, an atomic actor does not allow a “sub-model”. Each model inside of a hierarchical model is governed by a Model of Computation (MoC). A MoC, represented as a director on Fig. 2.1, controls the execution semantics of the model, defining how an actor behave and interact with other actors.

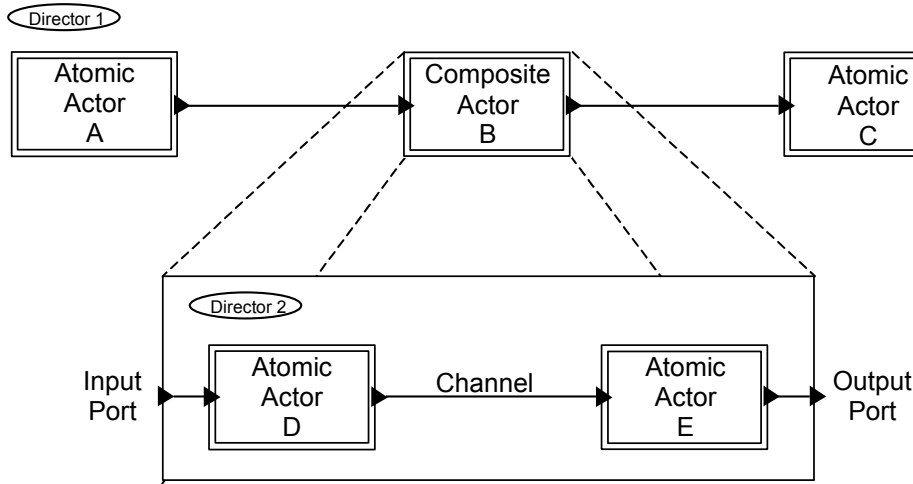


Fig. 2.1: Main components of actor-oriented models.

There is a rich variety of MoCs that deal with concurrency and time in different ways [2]. Some examples are: Component Interaction, Communication Sequential Processes, Continuous Time, Discrete Events, Distributed Discrete Events, Discrete Time, Finite-State Machines, Process Networks, Synchronous Dataflow, Giotto, Synchronous Reactive and Timed Multitasking. More information about different MoCs can be found on [2].

The Discrete Event (DE) MoC is of special interest for this work due to its popularity for simulating digital hardware, including simulators for VHDL and Verilog languages [2]. In the DE MoC, events generated by actors are enqueued in a global queue of pending events sorted by time stamp. When the event’s time stamp arrives, the DE simulator

dequeues the event and triggers the appropriate actor to execute.

There are many examples of actor-oriented languages, frameworks, and software techniques, including Simulink, Labview (National Instruments), Modelica (Linkoping), GME (Generic Modeling Environment from Vanderbilt), Easy5 (Boeing), SPW (Signal Processing Worksystem from Cadence), System Studio (Synopsys), ROOM (Real-time Object-Oriented Modeling from Rational), VHDL, Verilog, SystemC (various), Polis & Metropolis (UC Berkeley), and Ptolemy & Ptolemy II (UC Berkeley). Many of these, like Simulink, use a visual syntax to represent actor-oriented designs. Some are used for designing hardware, some for software, and some for both [3].

2.1.2 Ptolemy II

As presented on the previous Section, several frameworks use actor-orientation. Among them, Ptolemy II is an open-source alternative that allows heterogeneous models to be created by using different MoCs on each level of a hierarchical model. The communication among actors is managed by a director according to a given MoC and it is based on the exchange of data tokens through channels. The MoCs used as example on the previous Section are the directors accepted by Ptolemy II.

Ptolemy II is implemented in Java and contains a graphical user interface named Vergil, presented on Fig. 2.2. Vergil includes an extensive library of specialized components with different functionalities, ranging from basic flow control to complex signal processing. Shorter design times can be achieved through the appropriate use of those parameterizable components.

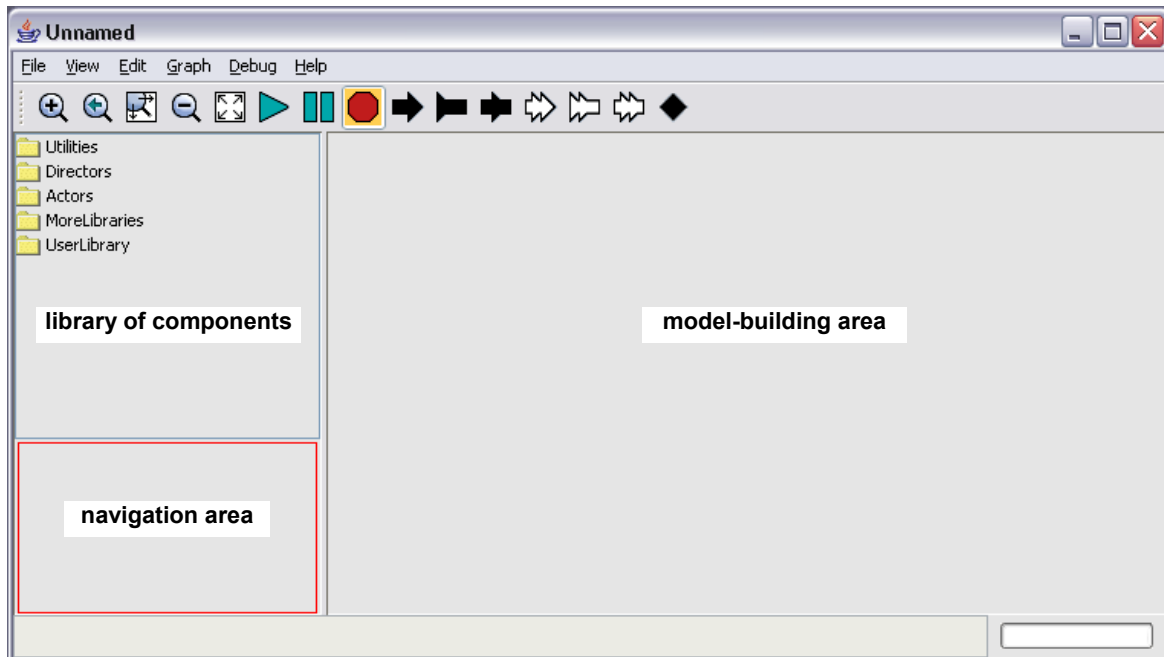


Fig. 2.2: Graphical interface from Ptolemy II called Vergil.

2.2 Communication Infrastructure Modeling

Taking into account the previously presented summary about actor-orientation and the Ptolemy II framework, the next Sections present the different communication infrastructures used on this work. Initially, related works in the area of communication infrastructure modeling are presented. Next, the HERMES NoC is presented as a reference for the NoC models created. Finally, three communication infrastructure models are illustrated.

2.2.1 Related Works

According to Bjerregaard et al. [4] modeling NoCs through abstract models is the first means to approach and understand the required NoC architecture and the impact of the traffic within it. In this context, some research groups try to adapt generic network simulators to the intra-chip environment, while others propose tools/techniques to specify, simulate and generate NoCs.

Generic network simulators used in the NoC context are the NS-2 [5][6] and the OPNET [7][8][9][10]. Both give support to the description of the network topology, the communication protocols, routing algorithms and traffic (*e.g.* random traffic). These network simulators do not consider some particularities inherent to on chip structures (*e.g.* communication bandwidth between routers), which can be very important to the design decisions.

Xu et al. [8] present an architecture-level methodology for modeling, analyzing, and designing different NoC architectures. Due to its low level abstraction, this approach can estimate with high accuracy the performance, power, and area of diverse NoC architectures. This methodology employs some available tools like OPNET, Design Compiler and SPICE, which requires long simulation times.

Bertozzi et al. [11] propose the NetChip synthesis flow, which allows the exploration of different NoCs topologies (such as mesh, torus, hypercube, Clos, and butterfly). The NetChip flow is composed by three phases: (i) NoC topology mapping, (ii) selection and (iii) generation (SystemC model that can be simulated at the cycle-accurate and signal accurate level). Additionally to the flow, an input core graph used in the mapping phase is captured based on statistical analysis and simulation.

Kogel et al. [12] propose a modular framework for system level exploration in TLM of the on-chip interconnection architecture. It allows capturing performance metrics like latency and throughput of different NoC configurations.

The OCCN framework proposed by [13] enables the creation of NoC architectures at different abstraction levels, protocol refinement, design exploration, and NoC component development and verification based on a communication API. The OCCN methodology has been adopted by Dumitrascu et al. [14] in order to analyze the effectiveness of inter-module communication and adaptation components. In this approach, the communication architecture performance evaluation is based on cycle-

accurate co-simulation.

Pestana et al. present in [15] a NoC simulator based on user-generated XML files that describe NoC topology, IP to NoC mapping and detailed interconnections. The simulator also allows describing traffic generators to evaluate NoCs.

Most of proposed techniques/tools allow the emulation of the NoC in different levels of abstraction, considering different architectures (e.g. topology, router) and traffic conditions. In many cases, the simulation occurs in TLM style, which demands less design and simulation time than Register Transfer Level (RTL) description.

2.2.2 NoC Infrastructures

A network-on-chip (NoC) is a potential and efficient infrastructure to handle MPSoC communication requirements. Scalability, energy efficiency, and support to globally asynchronous locally synchronous (GALS) paradigm justify the adoption of this approach [16][17][215]. However, the adoption of NoCs includes new challenges to the MPSoC design flow, such as choosing a suitable routing algorithm, NoC topology, buffering strategy, flow control scheme and power consumption techniques. And each of such choices will affect how efficiently a particular NoC handles the traffic generated by a given multiprocessing application.

Due to the vast design space alternatives that these challenges may impose to the final application and its required performance, the evaluation of NoCs become a mandatory step in the MPSoCs design flow. The evaluation of NoCs is required to establish a good compromise between the NoC architecture characteristics and the requirements of the given application. An example of this is to investigate the correlation between the packet length and buffer depth [16].

2.2.2.1 High Level Abstraction Modeling of NoCs

To accelerate the design space exploration of NoCs, high level abstraction modeling of the NoC is needed. The design exploration at register transfer level (RTL) does not provide the required support to the design space exploration of MPSoCs based on NoC communication architecture. The level of details that have to be modeled and the low accessibility and visibility of the components' behavior justify that affirmative.

The issues mentioned above, combined with time to market pressure, demand high abstraction level modeling and more appropriate debugging capabilities. The high level modeling activity is a trade-off between level of details and model confidence. In NoC context, the *level of details* refers to the structure and behavior abstraction of the NoCs' components. The *structural abstraction* refers to: (i) the granularity of data storage (e.g. storage for a flit or packet); (ii) the number of components that will be considered or abstracted and how they are interconnected (e.g. the number of wires or a channel). The *behavior* abstraction includes how, and more importantly, when such components (e.g. arbiter) update their internal state and concurrently interact with other components (e.g. buffer). The *model confidence* means how useful the model is for a particular purpose

regarding the accuracy results between the model and its reference scenario.

2.2.2.2 Top-Down NoC Modeling

One difficult scenario faced by designers is to interactively navigate through the wide design space of parameterizing a NoC for an MPSoC. Such scenario includes the system-level specification of an initial solution, the evaluation of design alternatives through formal methods and/or simulation and the refinement of the specification towards a reference design that can be used for hardware synthesis.

In practice, such ideal top-down scenario is not always possible. Most NoC models reuse previously designed modules such as arbiters or FIFO buffers, and such modules are usually implemented at RTL (Register Transfer Level) or logic levels using HDLs. Thus, even if parts of the interconnect architecture are designed or specified using more abstract models, the only reference model that describes the complete functionality of the interconnect is usually the one implemented using HDL. This makes the interactive design space navigation and the exploration of design alternatives more difficult, since HDL models simulate slowly, are hard to change and require additional expertise if they are to be validated with complex testbenches.

A common practice to facilitate the exploration of design alternatives is to build a simplified model that abstracts irrelevant parts of the RTL description of the NoC, allowing quick "what-if" experiments and providing a simple interface to verify its performance and functionality. However, the usefulness of abstract models depends on how well they can capture the original behavior described at RTL. The abstract modeling activity becomes a trade-off between precision (which is associated to complexity and slow simulation speeds) and abstraction (which, if excessive, renders the model useless).

The development of such abstract models must contemplate functional details such as internal latencies, congestion effects, routing and arbitration delays, so that designers can properly explore the NoC design space, optimizing the interconnect architecture to a particular traffic scenario.

Fig. 2.3 depicts the approach used in this work, highlighting the fact that the overhead due to the abstract modeling can be compensated by the potential improvement on the final design due to the extensive analysis and exploration of alternatives at a higher level of abstraction. It is worth noticing the two-way flow between the RTL model and its abstract counterpart: while the former serves as reference for the creation and validation of the latter, all optimization and design exploration activities are done at the abstract model, which is then used as reference for the creation of an optimized RTL model.

2.2.2.3 Ptolemy II for NoC Modeling

The choice of Ptolemy II for modeling a NoC is justified mainly due to following characteristics:

- its potential for abstracting structural elements of a NoC;
- its token-based communication across generic channels supporting data type inference;
- its potential to model behavioral elements;
- its heterogeneity, by applying different models of computation at different levels of the design hierarchy. This can be very useful to obtain cost and performance figures for a given NoC architecture considering different application scenarios while balancing the accuracy of the behavior of each model element regarding time and concurrency. For instance, for obtaining accurate figures of maximum buffer occupation in an interconnect it may be necessary to model it using cycle-accurate discrete events or, alternatively, following a synchronous dataflow execution semantics that can abstract time completely;
- its flexibility, by using both the library of parameterizable components and the set of directors as an object-oriented framework, thus with extensibility as a major goal. That way, Ptolemy II can be applied to different domains and follow different methodologies as originally envisioned. Examples include [18], which added a library of actors aimed specifically to wireless communication systems, and [19], which extended the set of directors to include the possibility of co-simulate UML sequence diagrams along with actor-oriented models.

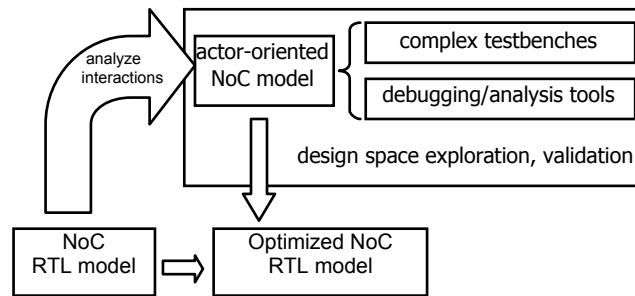


Fig. 2.3: Proposed approach for NoC modeling and design space exploration [102].

2.2.3 HERMES

HERMES [215] is a parameterizable NoC infrastructure specified in VHDL at RTL, aiming to implement low area overhead packet switching NoCs. The first version of HERMES was created on the bachelor work of the current author of this thesis and Aline Vieira de Mello under the supervision of professors Fernando Gehm Moraes and Ney Laert Vilar Calazans in 2003. Since 2003, the HERMES NoC has been receiving improvements from researchers around the world and a tool called Atlas was created to merge these improvements in a single compatible project. The functionalities of Atlas are not only to create, parameterize, simulate and debug the HERMES NoC, but also to generate traffic and evaluate the power and the performance of the NoC. Some parameters used to generate an HERMES are the flow control mechanism (e.g. handshake, credit based), topology (e.g. mesh, torus), NoC dimension, flit width, buffer depth, routing algorithm and number of virtual channels. The Atlas is distributed for

free on the Internet.

On HERMES all data is partitioned in packets, which are composed by a header and a payload, as illustrated by Fig. 2.4. A packet is transmitted on the network in units called flits (flow control digits), which are the smallest unit handled by the flow control. The flit width for the HERMES is parameterizable, and the maximum number of flits in a packet is $2^{(\text{flit width, in bits})}$. Packets are sent from a processing element to a given destination through routers that have some awareness of the interconnect topology and thus manage to deliver the packet while avoiding deadlocks and livelocks. Its routers have centralized switching control logic and five bi-directional ports. One port is used to establish the communication between a router and its local processing element, while the others are connected to the neighbor routers. Each input port stores received data on a FIFO buffer.

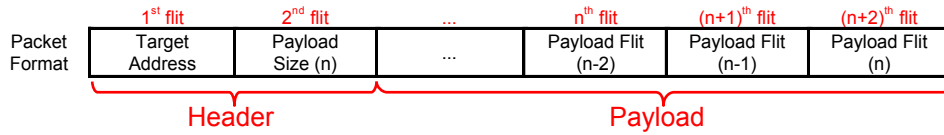


Fig. 2.4: Standard packet format accepted by HERMES NoC.

2.2.3.1 Switch control

The two main functionalities of the switch control are to implement the arbitration and routing of the received packets. In order to keep the area consumption of the NoC low, only one routing module exists for all 5 input ports of the router. Therefore, an arbiter is used to choose one input when more packets arrive at the same time.

The arbitration algorithm used is the Round Robin and it works as follows. The priority of a port is a function of the last port having a routing request granted. For example, if the local input port (index 4) was the last to have a routing request granted, the East port (index 0) will have greater priority, being followed by the ports West (index 1), North (index 2), South (index 3) and then Local (index 4) again. This method guarantees that all input requests will be eventually granted, preventing starvation to occur. The arbitration logic waits four clock cycles to treat a new routing request. This time is required for the switch to execute the routing algorithm. If a granted port fails to route the flit, the next input port requesting routing have its request granted, and the port having the routing request denied receives the lowest priority in the arbiter.

The routing algorithm used is the XY and it works as follows. After arbitration is done, the XY routing algorithm compares the actual switch address (xLyL) to the target switch address (xTyT) of the packet, stored in the header flit. Flits must be routed to the local port of the switch when the xLyL address of the actual switch is equal to the xTyT packet address. If this is not the case, the xT address is first compared to the xL (horizontal) address. Flits will be routed to the East port when $xL < xT$, to West when $xL > xT$ and if $xL = xT$ the header flit is already horizontally aligned. If this last condition is true, the yT (vertical) address is compared to the yL address. Flits will be routed to South when $yL < yT$, to North when $yL > yT$. If the chosen port is busy, the header flit as

well as all subsequent flits of this packet will be blocked. The routing request for this packet will remain active until a connection is established in some future execution of the procedure in this switch.

When the XY routing algorithm finds a free output port to use, the connection between the input port and the output port is established and the *in*, *out* and *busy* switching vectors at the switching table are updated. The *in* vector connects an input port to an output port. The *out* vector connects an output port to an input port. The *busy* vector is responsible to modify the output port state from free (0) to busy (1). Consider the North port (index 2) in Fig. 2.5. The output North port is busy (*busy*=1) and is being driven by the West port (*out*=1). The input North port is driving the South port (*in*=3). The switching table structure contains redundant information about connections, but this organization is useful to enhance the routing algorithm efficiency.

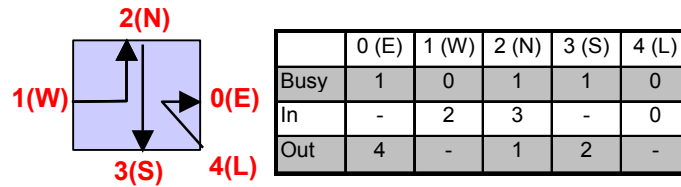


Fig. 2.5: Three simultaneous connections in the switch (left), and the respective switching table (right).

After all flits composing the packet have been transmitted, the connection must be closed. This could be done in two different ways: by a trailer or using flit counters. A trailer would require one or more flits to be used as packet trailer and additional logic to detect the trailer would be needed. To simplify the design, the switch has five counters, one for each output port. The counter of a specific port is initialized when the second flit of a packet arrives, indicating the number of flits composing the payload. The counter is decremented for each flit successfully sent. When the counter value reaches zero, the connection is closed and the *busy* vector corresponding position of the output port goes to zero (*busy*=0), thus closing the connection.

2.2.3.2 Crossbar

The crossbar is responsible to connect the outputs of the router to the inputs. This happens instantaneously after the routing table presented on Fig. 2.5 is written due to the combinational logic used on its implementation.

2.2.3.3 Buffer

The main responsibility of the buffer is to temporarily store flits of a packet in a FIFO (First In First Out) structure during arbitration, routing and congestions. In the case of HERMES, buffers are connected on the input ports of the routers. After executing arbitration and routing, as explained on Section 2.2.3.1, flits are forwarded to the output port of the router by using the crossbar explained on Section 2.2.3.2. As the output port of a router is directly connected to an input port of a neighbor router on the network, a buffer communicates directly with a neighbor buffer. This communication is called flow control.

One alternative of flow control is the handshake, which is built on top of the physical interface presented on Fig. 2.6, to deal with the correct transmission of flits. In this protocol, when the switch needs to send data to a neighbor switch, it puts the data in the *data_out* signal and asserts the *tx* signal high. Once the neighbor switch stores the data from the *data_in* signal, it asserts the *ack_rx* signal high, and the transmission is complete.

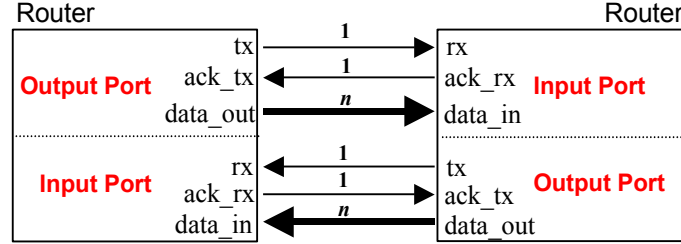


Fig. 2.6: Physical interface between routers.

2.2.4 RENATO

RENATO is an actor-oriented model based on the behavioral patterns of the HERMES NoC presented on Section 2.2.3. To better understand and abstract the relevant behavioral patterns in the RTL model of the NoC interconnect, each inter-component interaction was formalized using UML sequence diagrams [20]. Such approach was required after a number of failed attempts to simply convert the VHDL code into a more abstract model. In such attempts, the numerous implementation-related details present in the VHDL model were often preventing the proper capture of the conceptual behavior of the system.

By describing a system through its inter-component interactions, it was possible to isolate the individual functionality of each conceptual actor in the system, disregarding the fact that many of such actors had been merged in the VHDL implementation. For instance, the arbitration and routing procedures were implemented as a single state-machine in the VHDL model for the sake of simplicity and efficiency. However, on the abstract model, such procedures must be implemented separately in order to facilitate the design space exploration (e.g. to investigate the positive and negative impacts of using adaptive routing).

The behavioral patterns extracted from the interactions of the HERMES routers, represented by the UML sequence diagrams illustrated on Fig. 2.7, are the base for the RENATO NoC model. These UML sequence diagrams are the arbitration request by a particular input buffer (a) and the transmission of a flit from one input buffer to a neighbor router through an output channel (b) [102].

Interaction (a) in Fig. 2.7 illustrates the establishment of a connection between an input buffer to an output port. Initially, the input buffer sends the packet header to the routing controller. The routing controller asks the arbiter to choose one of the possible incoming requests that can arrive from any of the five input buffers. After getting a positive response, the arbiter sends the header flit to the router and the router executes a

particular algorithm (for instance XY algorithm) to determine which output port the packet should be sent to. Once the routing is done, the controller verifies if the chosen output port is free. If the output port is free, the input buffer establishes the connection to the output port, otherwise the connection is refused and the input buffer must start the whole input-output connection requesting process again.

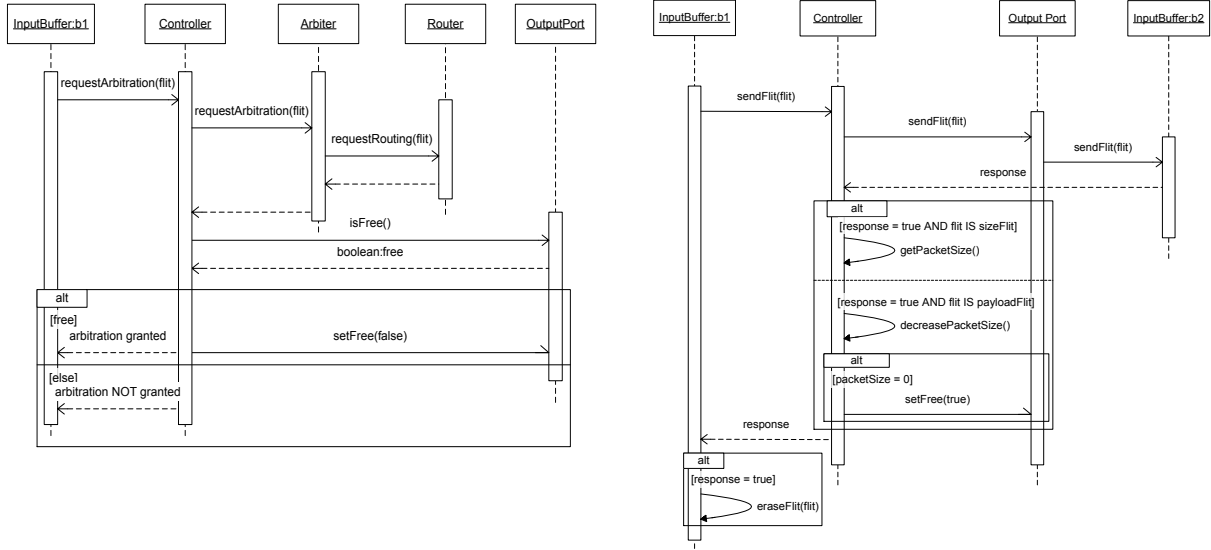


Fig. 2.7: UML sequence diagrams depicting interactions between components of HERMES NoC [102].

Interaction (b) in Fig. 2.7 models the transmission of a flit between two neighboring routers. The controller receives the flit from the local input buffer and checks which output port is allocated to it (the port allocation was done previously in interaction (a)). It then sends the flit to the remote input buffer through that port, and waits for an acknowledgement. The controller is configured to keep track of the different parts of a data packet, so it will read the packet size (if it is the second flit of a packet in the case of HERMES) or update its counter of sent flits (for all subsequent payload flits) upon receiving the acknowledgement from the remote input buffer. Finally, the controller notifies the local input buffer about the success of the flit transfer. If not successful, the input buffer will then repeat the whole interaction.

It is worth noticing that the sequence diagrams simply define a partial order between messages, without any particular reference to time and exposing potential concurrency. Such untimed model is enough to analyze the functionality of the interconnect, but it can't provide an accurate behavior of the system over time. For instance, it is not possible to determine the latency of a given packet or the occupation of a given input buffer.

To obtain accurate timing behavior, which in turn can provide metrics for proper design space exploration, this work explores the annotation of timing information for each of the transactions depicted on an UML interaction. Such timing information can be obtained from a cycle-accurate RTL simulation or can be a design-time estimate. In both cases, such models become a versatile tool for design space exploration, since it allows a

simple way to explore design alternatives. For example, the message *sendFlit* between the output port and the remote input buffer in interaction (b) can have alternative implementations using a handshake protocol or a credit-based flow control. By annotating the message with timing information from a cycle-accurate simulation, a designer can compare the impact of the use of each one of them (in the case of HERMES, a handshake takes twice as much time as a credit-based flow control). This timing annotation used on this example can be configured by simply clicking over the ‘delay’ component presented on the right side of Fig. 2.8 and setting how long does it take for the acknowledgement to arrive back to the neighbor router that was sending a flit.

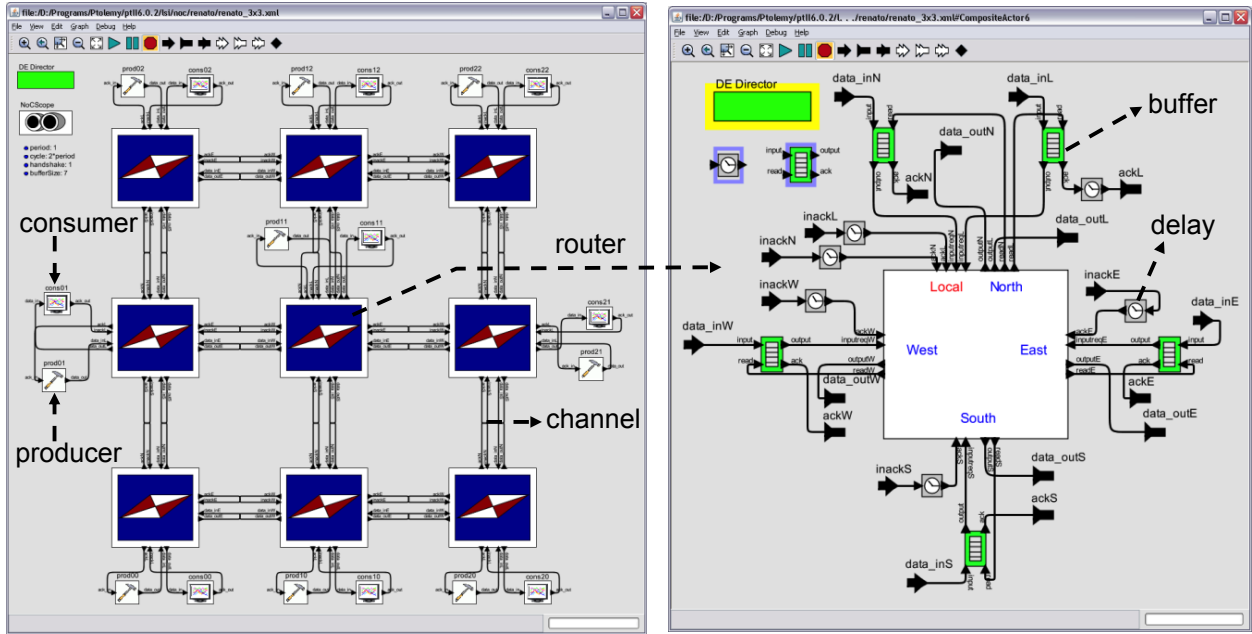


Fig. 2.8: Screenshot of a 3x3 RENATO NoC and a router.

Fig. 2.8 presents not only a 3x3 RENATO NoC modeled inside the Ptolemy II framework, but also the internal components of one router of the NoC. The NoC is composed by ‘routers’ interconnected by ‘channels’. One ‘channel’ is composed by four wires, providing exactly the same functionality as the HERMES interface depicted on Fig. 2.6 with six wires. RENATO uses fewer wires than HERMES because the *data_out* implicitly performs the behavior of the *tx* signal when a token is transmitted through the *data_out* signal. While the east, west, north and south ports of the router are connected to neighbor routers, the local port is connected to the IP core, which is represented here by ‘producers’ and ‘consumers’. Producers send packets to the NoC in form of tokens. Tokens experience the delay of the NoC, and after passing through possible congestions arrive to the target ‘consumer’. The ‘consumer’ is responsible to receive the token and perform what it was programmed to do (e.g. pass it to the IP core, write it on a file, count it, discard it). A router of the RENATO NoC, depicted on the right side of Fig. 2.8, is composed by ‘buffers’ connected to its input ports exactly as the HERMES NoC. The central block is the router core, which is implemented in Java and performs equivalent functionalities as the switch control and the crossbar of HERMES described on Section 2.2.3.1 and on Section 2.2.3.2.

The performance results of RENATO were compared to the results of HERMES (cycle-accurate simulation) using different traffic scenarios and NoC configurations generated by the Atlas framework, presented on Section 2.2.3. Atlas was also used to generate the different HERMES configurations and to perform the analysis of the results. Two different NoC topologies were evaluated, 3x3 and 4x4 meshes, both with routers containing 8-position input buffers, handshake flow control, centralized arbitration and XY routing algorithm. The generated traffic has a transmission rate of 200Mbps, random destination nodes and the interval between packets followed a normal distribution. Fig. 2.9 shows the obtained results when comparing the average latency of all packets (normalized to HERMES clock cycles) considering five distinct number of packets per producer.

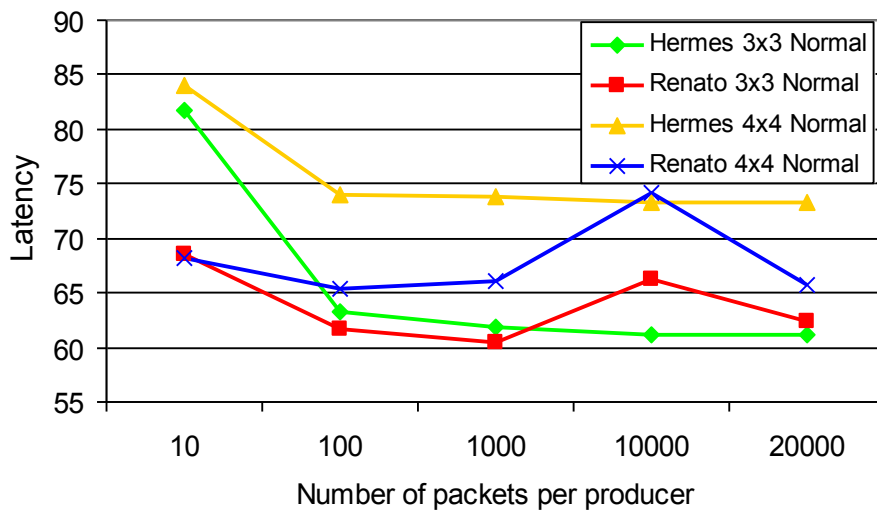


Fig. 2.9: Comparison of average latency obtained by HERMES RTL simulation and RENATO actor-oriented simulation [102].

Tab. 2.1 shows the error of the average latency of RENATO in comparison to HERMES, in percentage. For long-lasting traffics, the error is in the order of 10%, which is a very good figure considering that the actor-oriented model is based only on interactions and works without the synchronization of a clock signal. That is critical in systems like a NoC interconnect, where small differences can cause a significant difference on the overall performance. For instance, if the arbitration request of one input buffer arrives to the arbiter slightly late, the latency of that packet can be increased in hundreds of cycles.

2.2.5 JOSELITO

JOSELITO is a more abstract NoC model than the RENATO NoC model presented on the previous Section. However, it uses the same UML interactions defined in Fig. 2.7. The main difference between JOSELITO and RENATO is the decrease in JOSELITO's simulation time, caused by the reduction of the number of communication events caused by the flit by flit packet forwarding. This is accomplished by combining simulation and analytical methods, what is here called Payload Abstraction Technique (PAT).

Tab. 2.1: Average latency error between the models [102].

NoC size / Traffic	Number of packets per producer	Average Latency HERMES	Average Latency RENATO	Average Latency Error (%)
3x3 - T1	10	81.72	68.46	16.22
3x3 - T2	100	63.32	61.62	2.68
3x3 - T3	1000	61.85	60.51	2.16
3x3 - T4	10000	61.1	66.22	-8.37
3x3 - T5	20000	61.09	62.44	-2.20
4x4 - T1	10	84.04	68.23	18.81
4x4 - T2	100	73.96	65.42	11.54
4x4 - T3	1000	73.85	66.07	10.53
4x4 - T4	10000	73.31	74.17	-1.17
4x4 - T5	20000	73.33	65.73	10.36

The PAT comprises that: (i) the packet is defined as a *header* and a *trailer*, (ii) the buffer is a FIFO structure modeled as a finite state machine, (iii) packet headers are released from a given router once there is available buffer space at next hop on its route, (iv) and a simple analytical method is used to calculate the packet trailer release time [101].

Usually packets are composed by a header, a payload, and a trailer. In this technique, the packet structure is abstracted in *header* and *trailer*, as shown in the right bottom part of Fig. 2.10. This abstraction eliminates the *flit by flit* payload transfer.

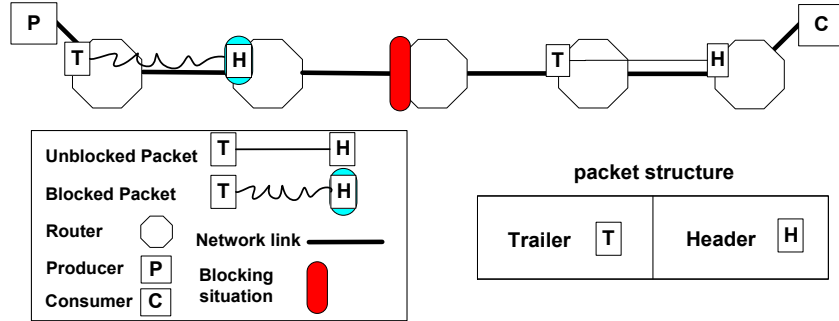


Fig. 2.10: Packet transmission situations (unblocked and blocked), and packet structure [101].

In this technique, the NoC routers are modeled using the entities *buffer* and *control*. The entity *control* is responsible for input port arbitration, routing algorithm, and input to output port data forwarding. The entity *buffer*, a FIFO structure, is modeled as a finite state machine (FSM), with four states: (i) *empty*, (ii) *header*, (iii) *waiting trailer*, and (iv) *trailer*.

In the *empty state*, the buffer is able to receive a packet header. The received header is stored into the buffer in the *header state*. After forwarding the packet header, the buffer goes to the *waiting trailer* state. When the trailer is received, the FSM goes to *trailer state*. Once the trailer is forwarded and removed from the buffer, it returns to the *empty state*.

To ensure correct functionality during packet transfers and to obtain high precision latency and throughput results, a sender (producer or router) releases the packet trailer according to an analytical method. One possible solution is presented in Eq. 2.1, which

indicates the packet trailer release time ($ptrt$) including when the header is forwarded (hft), the packet size and the number of cycles to transmit one flit (ctf) from one hop to another router or consumer. The ctf is expected to be one for credit-based and two for handshake control flow (clock cycles).

$$ptrt = hft + pcksize \times ctf \quad \text{Eq. 2.1}$$

The header forwarding time depends on the number of clock cycles required to execute the arbitration, the routing algorithm and the successful reception of the header by the neighbor resource (router or consumer). This parameter is obtained from the RTL NoC simulation. After reaching the packet trailer release time, defined by Eq. 2.1, the packet trailer is sent, following the same path reserved by the header.

When a header packet arrives in an input buffer, two blocking situations can occur: (i) the desired output port is reserved by the control entity to another input port; (ii) the target neighbor input buffer is in the trailer state. The packet trailer can be blocked only when the target buffer is in the header state. In this case, after sending the header, the packet trailer is forwarded after $ptrt$ clock cycles. The connection between an input and an output port of the router will be closed right after sending the packet trailer to the neighbor input buffer.

According the proposed technique, three transmission scenarios are possible. For the sake of simplicity, the following explanation assumes that the abstracted packet payload requires 4 input buffers (4 hops, including the header position, Fig. 2.11) in the path consumer-producer, i.e., the relationship between the real payload size and the real buffer depth is 4. This means that when the packet header arrives at the consumer, the trailer must be stored into the input buffer of the router located 4 hops before its consumer location. In the following examples, consider arbitration/routing requiring 7 clock cycles, payload size equal to 21 flits and credit-based flow control ($ctf=1$). The packet header (H) arrives at the first router (R1) at time 0.

Scenario (i): Blocking-free delivery

After sending the header, the producer (P) forwards the trailer after 21 cycles (Eq. 2.1). This is the best case scenario, without any resource conflicts, resulting in a blocking-free delivery. In this case, there is no loss of accuracy in the latency evaluation due to: (i) the trailer arrives at the consumer after 21 cycles after the header received time; (ii) the connection between P and R1 will be closed after 21 cycles of the header forwarded time (hft equal 7). After sending the header to the next router (R2) at cycle 7 (hft), the Eq. 2.1 is applied (R1) and the trailer is forwarded to the next router (R2) at the cycle 28 (Fig. 2.11).

According to [101], in a blocking-free delivery scenario, the latency of a packet ($pcklatency$) from producer (P) to consumer (C) is obtained from Eq. 2.2.

$$pcklatency = nhops \times arbt + pcksize \quad \text{Eq. 2.2}$$

Where $nhops$ is the number of hops between P and C, $arbt$ is the arbitration time and

$pcksize$ is the size of the packet measured on number of flits. Applying Eq. 2.2 using the parameters of the scenario (i), the same 56 clock cycles are obtained.

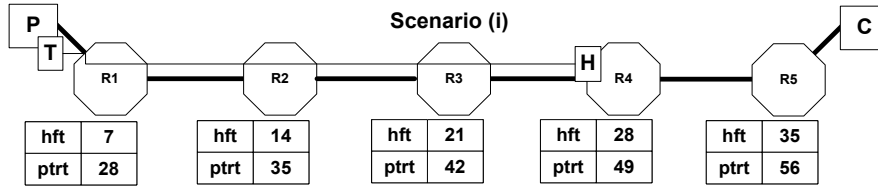


Fig. 2.11: Estimated release times regarding blocking-free delivery scenario [101].

Scenario (ii): Header Blocking

The header is sent by the producer, but in the fifth router (R5) a blocking situation is detected, as illustrated in Fig. 2.12. During the header blocking period (assuming 7 clock cycles), the trailer is forwarded two hops further (P to R2), decreasing the number of hops between the header and trailer (2 instead of 4 hops). Consequently, the connection between P and the first router (R1) is closed without considering the impact of the header blocking, which can lead to loss of accuracy on blocking other packets.

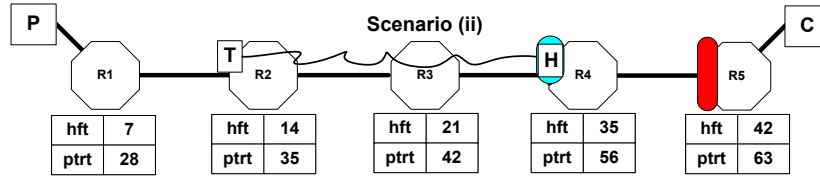


Fig. 2.12: Packet forwarding situation regarding header blocking [101].

Scenario (iii): Header and Trailer Blocking

The header and the trailer are sent as described in the previous scenarios. Due to the high blocking time (assuming 14 clock cycles) the trailer (R3) is blocked by its packet header (R4), as shown in Fig. 2.13. When the trailer is blocked by the header, the Eq. 2.1 is applied again and the router R4 will release the packet trailer after $ptrt$ clock cycles of the header forwarded time (hft equal 42). In this case, the distance between the header and the trailer is just one hop. Thus, two connections (P to R1 and R1 to R2) are released without considering the impact of the header blocking, increasing the possibility of loss of accuracy on blocking other packets.

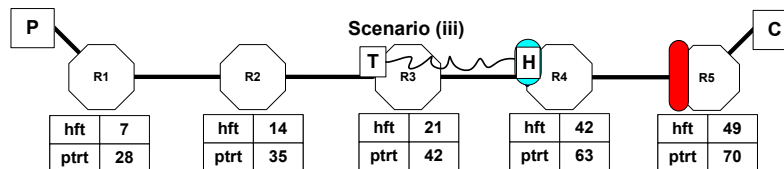


Fig. 2.13: Packet forwarding situation regarding header and trailer blocking [101].

It should be clear that the proposed technique is flexible in terms of modification and it is not restricted to the abstractions presented here. For example, different parameters can be added in the Eq. 2.1, improving the quality of the technique.

Two models are used as reference in the current work: an RTL-VHDL (HERMES - presented on Section 2.2.3) and a high abstraction model (RENATO - presented on Section 2.2.4). The first is used as reference for latency and throughput performance figures, since it is cycle accurate; the second is used as reference for simulation time, to enable the comparison between two models at a similar abstraction level. It is important to mention that JOSELITO and RENATO are back annotated with timing information from HERMES, thus allowing latency comparison among them. In order to make such analysis, the NoC models (HERMES, RENATO and JOSELITO) are evaluated varying:

- NoC sizes: 2x2, 3x3, and 4x4;
- traffic distribution: uniform (200 Mbps), normal (minimal rate 150Mbps, maximal rate 250Mbps, and standard deviation 10Mbps), and Pareto on-off (200 Mbps, maximum number of bursts set to 10 packets);
- number of transmitted packets per producer: 100 packets (T1), 1000 packets (T2), 10000 packets (T3), and 20000 packets (T4);
- packet size: 16 and 50 flits.

Fig. 2.14 presents the average latency simulation error (measured in clock cycles) of JOSELITO in comparison to HERMES for 3 different traffic distributions, 3 different NoC sizes and 16 flits per packet. The closer the lines are from the zero latency error, more similar JOSELITO and HERMES latency are obtained. The worst case average latency error presented is 3.4 clock cycles (Fig. 2.14(c) - 4x4 - 20000 packets). In this specific worst case, JOSELITO average latency is 5.26% lower than the reference model (according to the average absolute latency results presented in Tab. 2.2, 64.59 and 61.19 clock cycles for HERMES and JOSELITO, respectively).

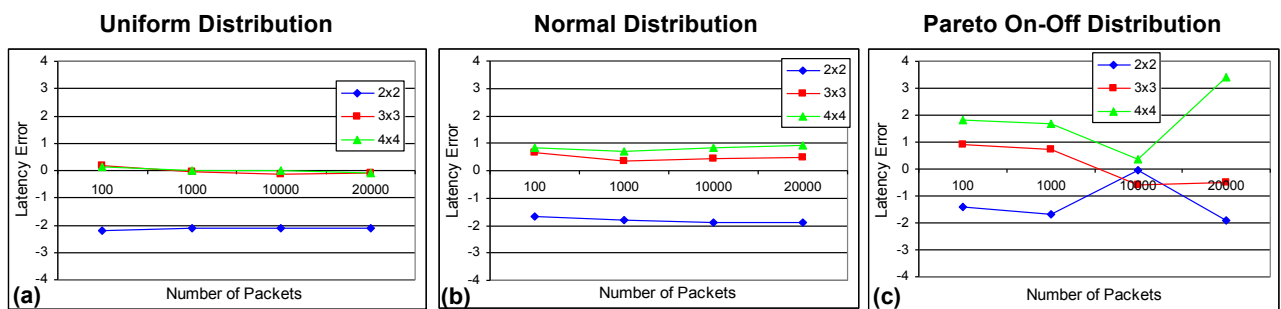


Fig. 2.14: Latency error between JOSELITO and HERMES (the RTL reference model) for 3 different traffic distributions (uniform, normal, and pareto on-off) and NoC sizes (2x2, 3x3 and 4x4). 16 flits packets [101].

Fig. 2.15 presents the JOSELITO latency error in comparison with HERMES when applying packets with 50 flits inside the NoC model. As the model does not yet consider the buffer size and it is calibrated to 16 flits per packet, the worst case error presented by 50 flits is 4.26 clock cycles (Fig. 2.15(b), 100 packets) in average. In this specific worst case, the absolute average latency to deliver all packets is 164.04 clock cycles in JOSELITO and 168.30 in HERMES. This represents only 2.53% latency error in comparison with the same case study in the reference RTL-VHDL model. It is important

to remember that the packet size is usually chosen by network interfaces, which is outside the NoC model and can usually be calibrated to any possible size.

Tab. 2.2 presents the absolute average latency and throughput simulation results for HERMES and JOSELITO using 16-flit packets. The worst case throughput error of JOSELITO in comparison to HERMES is 0.1% when sending 10000 packets per producer in a 4x4 Pareto on-off traffic distribution, which reflects multimedia applications behavior. These similar throughput and latency results between both models show that a high-level abstraction model can indeed replicate the behavior of a cycle accurate model.

Tab. 2.2: Average latency (“L” in clock cycles) and throughput (“T” in percentage of the relative channel bandwidth) for HERMES and JOSELITO, using 16 flits packets [101].

		Uniform Distribution						Normal Distribution						Pareto On-Off Distribution					
		2x2		3x3		4x4		2x2		3x3		4x4		2x2		3x3		4x4	
		H	J	H	J	H	J	H	J	H	J	H	J	H	J	H	J	H	J
T1	L	58.81	60.99	70.99	70.83	87.68	87.54	52.44	54.12	63.32	62.64	73.96	73.14	49.46	50.86	55.35	54.44	62.62	60.82
	T	12.88	12.86	7.21	7.17	4.37	4.34	13.95	13.99	7.28	7.34	4.37	4.38	12.04	12.03	5.30	5.31	3.16	3.16
T2	L	58.86	60.98	70.75	70.81	85.97	85.96	51.28	53.07	61.85	61.50	73.85	73.14	49.63	51.32	55.96	55.24	62.49	60.80
	T	13.76	13.77	7.29	7.28	4.55	4.57	13.41	13.42	7.13	7.13	4.44	4.45	13.25	13.27	6.80	6.81	4.32	4.32
T3	L	58.93	61.05	71.09	71.22	85.08	85.07	51.65	53.56	61.10	60.66	73.31	72.47	50.66	50.69	54.71	55.30	61.59	61.22
	T	13.69	13.69	7.20	7.20	4.53	4.53	13.4	13.40	7.06	7.06	4.42	4.42	13.39	13.44	7.12	7.11	4.50	4.40
T4	L	58.87	61.00	71.22	71.30	84.93	85.00	51.99	53.88	61.09	60.62	73.33	72.42	48.71	50.60	54.92	55.44	64.59	61.19
	T	13.69	13.69	7.25	7.25	4.51	4.52	13.47	13.48	7.09	7.09	4.45	4.46	13.53	13.45	7.02	7.08	4.41	4.41

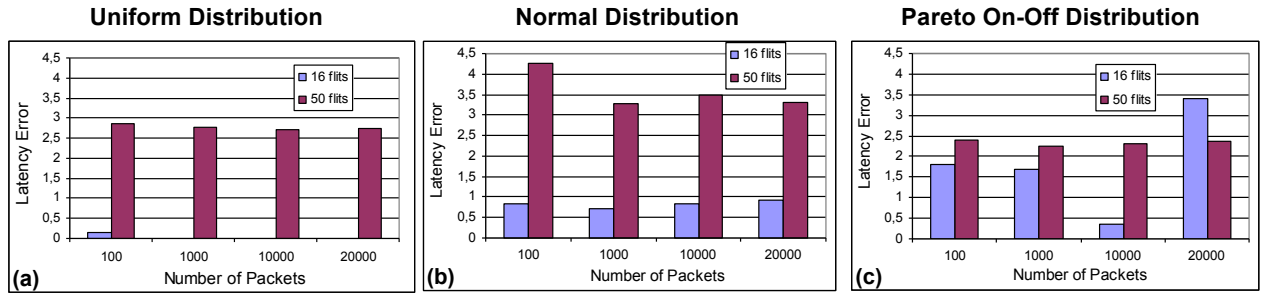


Fig. 2.15: Latency error between a 4x4 JOSELITO and a 4x4 HERMES for 3 different traffic distributions (uniform, normal and pareto on-off) and 2 different packet sizes (16 and 50 flits) [101].

Tab. 2.3 shows that JOSELITO is in average 2.3 times faster than RENATO in 88% of the executed case studies. These improvements in simulation time were achieved only by reducing the number of communication events originally caused by the flit by flit forwarding.

2.2.6 BOÇA

The third platform model used in this paper is the fastest and most abstract one, but it obviously pays for that by having the lowest accuracy. BOÇA considers the multi-hop nature of NoC communications and the buffering of flits at the input ports of each router, but it abstracts completely the arbitration logic and internal state of the routers.

Therefore, BOÇA's way to handle the interference among traffic flows does not reflect the real implementation.

Tab. 2.3: Speed up of JOSELITO in comparison to RENATO [101].

	Uniform			Normal			Pareto On-Off		
	2x2	3x3	4x4	2x2	3x3	4x4	2x2	3x3	4x4
T1	2.51	2.41	2.93	5.95	2.00	2.08	2.45	2.34	2.34
T2	2.52	2.09	2.27	2.81	2.01	2.01	2.54	1.52	1.64
T3	2.53	1.78	1.99	1.82	1.39	1.67	1.78	0.72	0.74
T4	4.07	1.38	1.68	1.70	1.14	1.67	4.71	0.60	0.70

2.2.7 Comparison of Platform Models

Fig. 2.16 illustrates the sequence diagrams of the application model used in this case study. The application is an autonomous vehicle that is modeled in five different sequence diagrams: *photogrammetry*, *obstacle recognition*, *direction adjustment*, *tyre pressure adjustment*, and *snapshot request*. The vehicle has two cameras, which capture images on the direction the vehicle is moving. The photogrammetry logic of the system pre-processes the images and extracts three-dimensional features by exploring the stereoscopy in both images. The obstacle recognition extracts the coordinates of the possible obstacles that might force the vehicle to adjust its direction. Obstacle coordinates are then fed to the obstacle database, which is also adjusted with the information from the ultrasonic sensor (which can measure more precisely the distance of obstacles that are closer). Direction adjustment logic can determine the vehicle's current position using GPS and then adjust the direction according the information of the obstacle database in order to avoid collisions with obstacles. The vehicle contains also sensors able to measure vibration. If the vehicle vibrates too much, it affects the quality of the camera images. Therefore, the vehicle is able to adjust the tyre pressure so that it is suitable for the surfaces it moves on. Finally, a radio interface enables interaction with external entities. In this example, the radio interface is only used to command the capture of images.

The application was simulated on the three different platform models, RENATO, JOSELITO, and BOÇA, all of them implementing a 4x4 mesh topology. Packets, that the application actors send to each other, have a maximum size of 48 flits, each flit of 16 bits. Empirically this packet size seems to be a good trade-off between the overhead the multiple packet headers generate versus the long-term occupation of platform resources, such as channels and buffers. Application messages that are larger than $48 * 16$ bits are divided in multiple packets.

Two different random mappings were used for each platform and the network latency for each message of each sequence diagram was measured. The network latency from the point the processing element sends a packet containing the message to the point the packet arrives at the processing element of the target node was also measured. That is, how long a packet spends in the network, either routed between switches or in an input buffer of a switch waiting for routing. Therefore, the latency of messages depends on

network congestion, data size of the corresponding packet, and the mapping (that is, if the sending and receiving actors of the packet are mapped to nodes that are close to each other or not).

The operation frequency of the platforms is set to 50 MHz and the systems were simulated for 18 seconds of wall clock time. Application-specific constraints define the execution frequency of each sequence diagram. Photogrammetry, obstacle recognition, and direction adjustment are executed once every two seconds, while tyre pressure adjustment and snapshot request once a second.

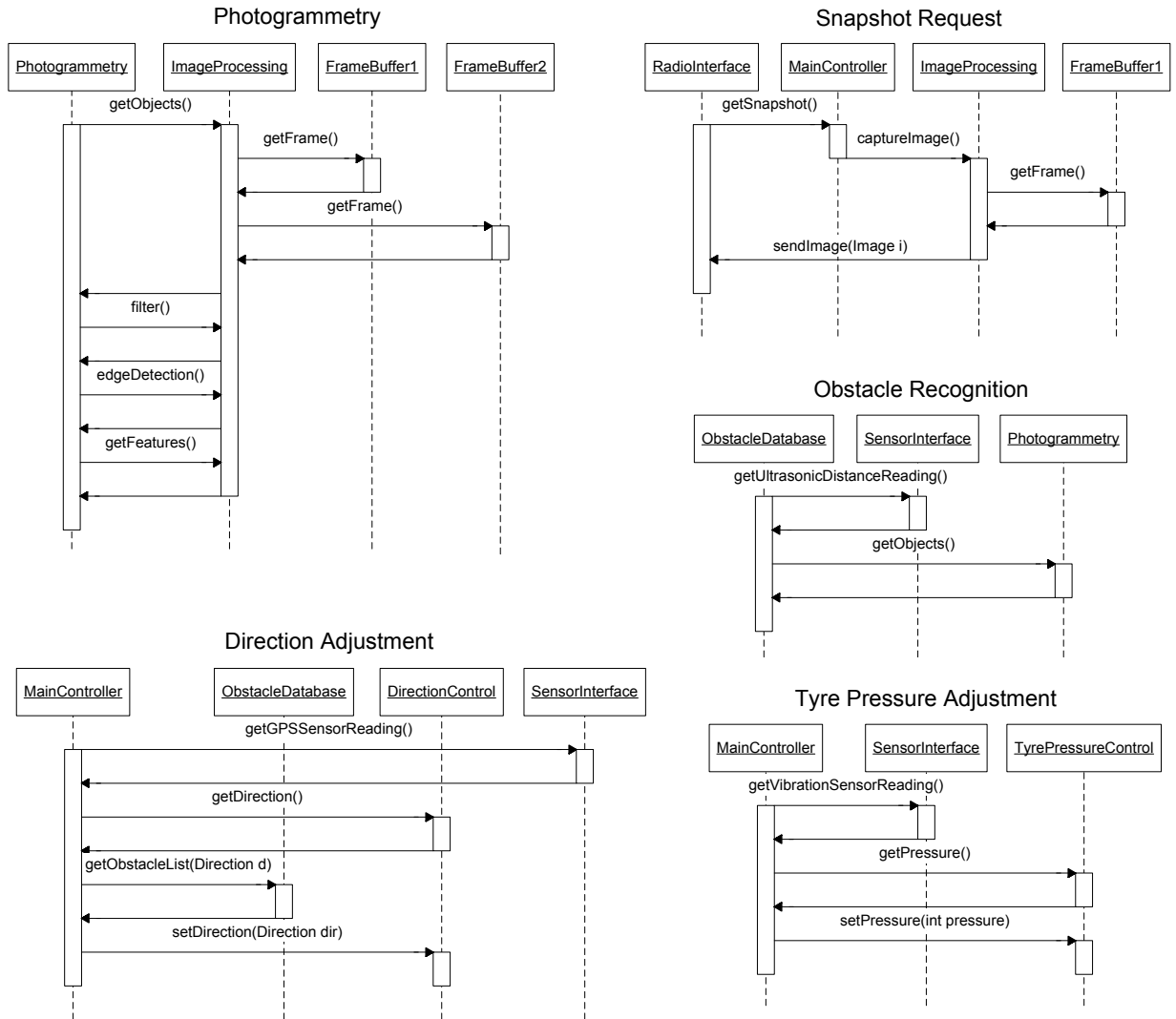


Fig. 2.16: UML sequence diagrams of an autonomous vehicle application [112].

Tab. 2.4 presents the worst case latency results for two different mappings of the application over RENATO, JOSELITO, and BOÇA. Since RENATO is the closest model to a RTL implementation, it is used as reference within this analysis. As expected, BOÇA presents a significant error for the worst case latency (around 45% in comparison with RENATO). This is due to the fact that it is not back annotated with timing delays from a real implementation and its modeling of network arbitration is very simplistic.

However, the simulation of the application over BOÇA was 402 times faster for mapping 1 and 492 times faster for mapping 2 in comparison with RENATO.

As soon as the application was executed on BOÇA, JOSELITO was used to extract more information about which mapping would work better when considering a mesh topology network. Even though JOSELITO has a high worst case latency error (around 30%), JOSELITO was proved to be useful, because every time JOSELITO has lower latency, RENATO has lower latency as well. The simulation of the application on JOSELITO was 2.8 times faster for mapping 1 and three times faster for mapping 2 in comparison with RENATO.

Tab. 2.4: Worst case latency results of two different mappings of the same application mapped onto three different platform models (error percentage compared to the reference model RENATO) [112].

	Mapping 1 (ms)			Mapping 2 (ms)		
	RENATO	JOSELITO	BOÇA	RENATO	JOSELITO	BOÇA
Direction Adjustment	0,39	0,27 (30%)	0,24 (37%)	0,38	0,26 (30%)	0,25 (32%)
Obstacle Recognition	0,12	0,09 (29%)	0,07 (41%)	0,10	0,07 (29%)	0,07 (29%)
Photogrammetry	1,41	0,99 (30%)	0,54 (62%)	1,37	0,95 (31%)	0,52 (61%)
Snapshot Request	1,35	0,94 (30%)	0,90 (33%)	1,31	0,90 (31%)	0,89 (32%)
Tyre Pressure Adjustment	0,14	0,09 (32%)	0,01 (55%)	0,11	0,08 (30%)	0,05 (50%)
Total	3,41	2,38 (30%)	1,76 (46%)	3,27	2,26 (31%)	1,78 (45%)

One additional observation we can make from Tab. 2.4 is related to long worst case latency for photogrammetry and snapshot request sequence diagrams. These long latencies are due to the large size of some of their messages. Only those two sequence diagrams are involved in transferring image frames from one processing element to another.

An important issue when modeling systems in different abstraction levels is the relative order of the platform models when exploring the design space of different features, such as mapping. Tab. 2.4 shows the relative ranking of RENATO, JOSELITO, and BOÇA for the two mappings. The same order is observed in both mappings. These results point out the fidelity among different platform models, enabling design space exploration at higher abstraction levels.

2.3 Monitoring and Debugging for NoC models

In the MPSoC context, software engineers have to deal with concurrency issues inherent to applications that require multiprocessing [21]. Thus, more appropriate debugging capabilities to design MPSoCs based on NoCs are demanded in order to increase and simplify the development of these systems.

Tracing parallel flows of communication is one critical challenging task of MPSoCs based on NoCs. The objectives of tracing communications are usually either debugging the NoC or monitoring the NoC for design space exploration. On Register Transfer Level (RTL) NoCs the tracing is frequently verified by waveforms, which are extremely complex due to the size of the NoC and provide limited useful information about the

global status of the NoC. Additionally, NoCs demand the analysis of a number of different signals from different routers and at different moments for tracing communication flows.

One disadvantage of NoCs when compared to busses, is the high complexity for monitoring the communication infrastructure. While it is easy to probe a bus and infer what the system is doing at one moment in time by listening the communication on the bus, a NoC is much more complex to infer due to the high number of wires to probe, the high number of parallel communications occurring at the same time and the different possible paths that two cores can use to communicate with each other. All this complexity to monitor NoCs only slows down the development and improvement of several NoC research works.

A number of works were found that explicitly address the intra-chip monitoring of NoCs. Marescaux et al. [22] present buffer monitors to detect congestions in the NoC and rapidly counteract. Once congestion is detected and the source of congestion found, a notification is sent over a separate control network to the source of congestion and the traffic is re-shaped.

Yin et al [23] propose a hierarchical monitoring approach to avoid sending all monitored information to a central point in the system, and thus avoiding the creation of a communication bottleneck. The monitors can be used for instance to measure throughput and power consumption.

People at the Eindhoven University of Technology and Philips Research Laboratories have been working on NoC monitoring since 2004 [24]. In their work an automated design flow is able to generate the Aethereal NoC, Silicon Hive processing cores, and transaction monitors. Transaction monitors are responsible to trace communications and are used in their case study for debugging purposes [25].

The previous works presented different purposes for using intra-chip monitors. Our purposes are design space exploration and debugging. However the method used here for debugging is different from works [24] and [25] because we employ a computer executable model of a SoC that uses NoC as a communication infrastructure, and not a prototype. Comparisons regarding the accuracy of this model with real hardware applications was presented on Section 2.2.

To improve the potential of observing and debugging the execution of an application running on top of a NoC architecture modeled according to the proposed approach, three different levels of debugging are provided:

- code-level debugging – this level supports the debugging of the internal functionality of the individual actors implemented within Ptolemy II. It can be done with regular programming code debugging tools like Eclipse, and is mainly used to verify if the sequential algorithm within a particular actor (for instance, the routing algorithm) works properly;
- interaction-level debugging – this level keeps track of the progress of the

interactions (e.g. as illustrated on Fig. 2.7). It uses textual output to inform the sending and receiving of each of the messages denoted in the sequence diagrams along with their timing information. To avoid overflow of information, it is possible to track the activity of any individual actors (for instance, observing only the input buffer of the direction “north” of the router with XY address “11” on a mesh);

- system-level debugging – a number of extended observability resources were implemented within the context of this work. They are extensions to Ptolemy II and can be plugged in different parts of the NoC model, working as probes which collect data from the network, process and display it graphically. All those extensions were combined on a small framework called NoCScope.

The NoCScope proposes the use of scopes to observe what is happening in the RENATO NoC model. A scope in this context is a special purpose graphical display to present information sent by monitors. Monitors are attached to NoC routers and they are responsible to compute data sniffed by probes. Scopes, monitors, probes and the whole RENATO NoC is developed in Java language. However, while the communication between routers follows the method calls and the concept of time provided by Ptolemy II, all monitoring related communications are implemented by native method calls in Java, therefore no extra traffic is generated in the NoC. In other words, the monitoring is transparent and not intrusive.

These monitors help the embedded software developers to quickly experiment new ideas and algorithms while having a good observability of what is happening in the NoC. The observability is improved by attaching seven different types of monitors to the RENATO NoC model. The information provided by the monitors may be used to improve several characteristics of the NoC, e.g. routing algorithm, arbitration algorithm, network topology, buffering strategy, flow control scheme, packet length, flit size, buffer depth and others.

Fig. 2.17 presents a screenshot of the seven monitors provided by NoCScope. Each scope will be explained in the following Sections.

2.3.1 PointToPointScope

The PointToPointScope (Fig. 2.17(a)) presents real time information about the NoC links in use and the input to output connections established by each router of the NoC. Therefore, it is possible to visualize the complete path that the packets are using inside the NoC. In this scope, each square represents one router. Each router can have an arrow, which represents that a connection is established between an input port and an output port. The arrow point to the output port, and near the end of the arrow a number is presented. This number is the target address of the communication. Arrows coming and going from/to the upper right part of the square are communicating with the local port (where a module is connected). This scope also presents a dot somewhere inside the square. This dot indicates the next input port that will receive the chance to connect to an output port (arbitration algorithm). At specific moments in time, it is possible to

notice that an arrow of one router does not match with an arrow in the next router (e.g. router 12, arrow from local port to south port). This happens because the neighbor router (e.g. router 11) did not have the chance to arbitrate and/or route the north port yet. Other times, an arrow is starting from nowhere (e.g. router 10, arrow from west port to north port), this is the end of a packet that is being transmitted. In this scope, the communication between one specific source of packets and one specific target receives one color. This happens to avoid confusion, as can be verified from arrows that are connecting router 00 to 02. If there were no colors, one would think that they belong to the same communication, when actually it is not the case. This scope works only in real time mode. The **real time mode** presents the current status of the system. This mode is recommended when the system is executed in a step by step way.

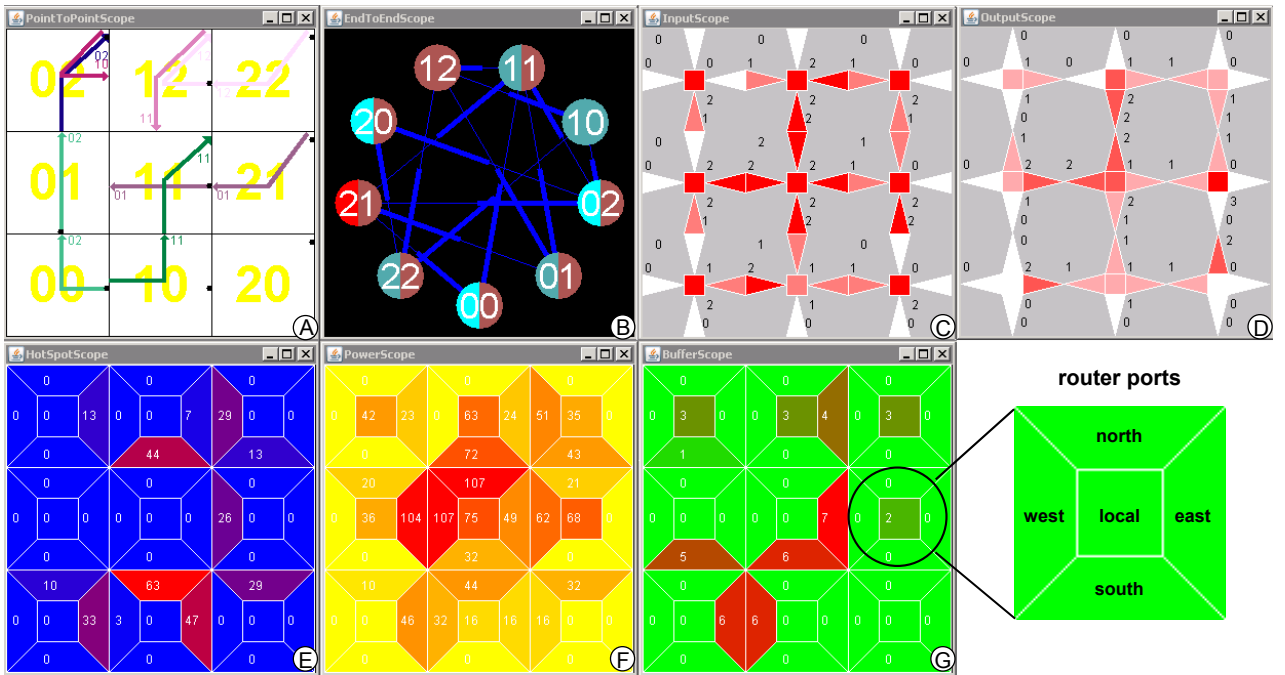


Fig. 2.17: Screenshot of the scopes used by the HermesDebugger.

2.3.2 EndToEndScope

The EndToEndScope (Fig. 2.17(b)) gives an overview of which module is communicating with which other module without considering the path of communication. Each circle represents one module. Thick lines mean that data is being sent by the module near the thick line. Narrow lines mean that data is being received by the module near the narrow line. If both sides of the line are thick, then both modules are sending data to one another. Circles are divided in half. Left half means input data coming in and right half means output data going out. Each half can change from color cyan (which means no data communication) to red (which means that this is one of the routers that are communicating most in the system). The EndToEndScope can display information in two different modes: real time and sliding window. The **sliding window mode** presents the accumulated status of the system in the last T simulation cycles, where T is the size of the sliding window measured in number of simulation cycles. This

mode is recommended when a continuous execution of the system is performed, because using the real time mode in a continuous execution may lead to miss short and fast events (e.g. the transmission of a single packet that rarely occurs between one source module and one target module).

2.3.3 InputScope

The InputScope (Fig. 2.17(c)) captures the activity of each input port of the NoC routers. This scope is useful when one is analyzing the most used ports and paths in the system. This scope works in sliding window mode or in accumulator mode. The **accumulator mode** counts the total amount of events, which can be usually configured as the total amount of packets or flits.

2.3.4 OutputScope

The OutputScope (Fig. 2.17(d)) works similar to the InputScope, but it measures the output ports of the NoC routers.

2.3.5 HotspotScope

The HotspotScope (Fig. 2.17(e)) emphasizes and quantifies the output ports of the NoC routers that are trying to send packets ahead to a neighbor router, but are being blocked due to congestion. On each simulation cycle that an output port of a router tries to forward a packet and the communication is denied, one unit is incremented on a counter that is displayed in the scope. The counter with biggest number in the NoC serves as basis to red color that is the place where the system is more congested. When little or no congestion is found, the color blue is displayed on the output port of the router. This scope is useful when one is analyzing the congestion of the system to try to counteract with e.g. a new placement of the modules, new schedule of tasks, usage of QoS techniques, usage of priority packets, usage of a different switching technique, reconfiguration of the buffer in execution time, usage of different physical or virtual channels, usage of a new reconfigurable channel. This scope works only in sliding window mode.

2.3.6 PowerScope

With the PowerScope (Fig. 2.17(f)) it is possible to obtain the power consumption of the routers by analyzing the transition activity on the channels. For example, let's assume that a channel between two routers has 8 bits of data, and the first binary value is "01000000" and the second is "00000100". In this case two transitions occurred, one in the second bit from 1 to 0 and one in the sixth bit from 0 to 1. More information about analyzing power consumption by measuring the activity on the channels can be found in [26]. In this scope the color yellow is used as little power consumed and red as much power consumed. This scope works in sliding window mode or in accumulator mode for total switching activity.

2.3.7 BufferScope

The BufferScope (Fig. 2.17(g)) measures the buffer occupation of the NoC routers. It is useful to calibrate the buffer size and find the trade-off between the total execution of an application (or maximum packet latency) and the amount of intra-chip memory to use. In this scope the color green is used as buffer empty and red as buffer full. This scope works only in real time mode.

2.4 Monitoring and Debugging for RLT NoCs

The previous Section presented the NoCScope monitoring system attached to the RENATO NoC model. This Section presents the connection of the NoCScope to the HERMES RTL NoC. Besides bringing all the advantages of a high-level debugging tool to a RTL NoC, a global picture of what is happening in the NoC and improved tracing capabilities are provided. This is accomplished by using a Java tool to represent graphically relevant events of the NoC. This tool is called HermesDebugger and requires as input a list of relevant events acquired from NoC router signals during the RTL simulation of the MPSoC. HermesDebugger interprets this list of events and displays meaningful information on the seven scopes presented on Section 2.3.

The most desirable solution would be to visualize the NoCScope information at the same time the HERMES NoC is being simulated, however no method was found on ModelSim simulator to send data directly to a Java code. ModelSim allows exporting values of certain chosen signals during the simulation in three different ways. The first one is to use the native VHDL log library to force the simulator to write the values of the signals in a log file every time some process is activated. However this solution required the modification of source NoC files. Second alternative is to write TCL codes that get the value of the signals during the simulation and export them. This solution was not used in favor of a third solution, that besides simpler, required few modifications in the project. This third solution is the use of the ModelSim log feature, which makes all results from a simulation accessible by ModelSim commands.

There are four ways of using the log information: through log files (*wlf* or *vcd* files), waveforms (*waves*) or lists (*lst* files). The *wlf* and *wave* files are better accessed using the ModelSim graphical tool. Only *vcd* and *lst* files can be easily read by text analysis.

The first choice was naturally the *vcd* (Value Change Dump) file. This is an ASCII-based format for dump files that defines names for each signal and, next, prints the changes in each signal in a time-ordered sequence. The advantages of this format are an output file with a great readability and low file size. However, because of a restriction in the ModelSim simulator, this file could only deal with *bit*, *bit_vector*, *std_logic* or *std_logic_vector* signals. Other types of signals like memory were silently ignored.

The next option is the list file (*lst* file), which consists of a file with the name and values of all chosen signals to be printed at all timestamps of the simulation. The problem lies in the size of the output file, which for long simulation periods could reach hundreds of

megabytes. This would demand a high volume of free memory to be read by a Java program.

Finally, the chosen solution was the use of an event list file. The structure is the same of a list file, but signals that do not suffer any modification in a certain timestamp are omitted. In fact, the event list file resembles the *vcd* file, but it is not so clean and organized. On the other hand, it works with all types of signals. An example of this file is presented in Fig. 2.18.

```

@0 +3
/TOPNOC/NOC/ROUTER0001/DATA_IN {0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000}
/TOPNOC/NOC/ROUTER0001/RX 00000
/TOPNOC/NOC/ROUTER0001/SWITCHCONTROL/FREE 11111
/TOPNOC/NOC/ROUTER0001/CREDIT_o 11001
@20 +3
/TOPNOC/NOC/ROUTER0001/DATA_IN {0000000000010000 0000000000000000
0000000000000000 0000000000000000 0000000000000000}
/TOPNOC/NOC/ROUTER0001/RX 10000

```

Fig. 2.18: Event list file example.

Lines starting with the symbol @ present the current simulation time and after the symbol + a delta time is presented. ModelSim uses delta times when many events need to be processed in the same simulation time, which may happen when some signal is sensitive to another signal that is changed in the current simulation time. All lines that follow a line which starts with symbol @ represent the complete name of a signal and its new value received in the current simulation and delta time.

The important VHDL signals required from the HERMES NoC to be exported by ModelSim and interpreted by HermesDebugger are the following:

- *buf*: temporarily stores all flits that have not been forwarded by the router yet
- *first*: pointer that indicates the position of the first flit stored in the buffer
- *last*: pointer that indicates the position of the last flit stored in the buffer
- *rx*: signal that indicates there is a flit in the input port waiting to be stored in the buffer
- *tx*: indicates a flit in the output port waiting to be sent
- *credit_o*: this signal indicates if the input buffer associated to this port still has space left to store an incoming flit (available only for credit based HERMES NoC)
- *ack_rx*: this signal indicates if the receiver accepted the flit received by this port (available only for handshake HERMES NoC)
- *data_in*: flit received by the input port of a router
- *busy*: indicates if an output port is occupied
- *mux_in*: indicates the output port connected to a specified input port

- *sel*: indicates the input port selected to send a packet
- *targetx*: stores the X coordinate of the target router from the received packet
- *targety*: stores the Y coordinate of the target router from the received packet
- *localx*: stores the X coordinate of the current router
- *localy*: stores the Y coordinate of the current router

The main task of HermesDebugger is to interpret the changes of the NoC signals through time and update every scope of the NoCScope accordingly. The main control window of HermesDebugger, presented in Fig. 2.19, allows the user to choose basic parameters, such as location of the list file, start time, step time, window size and speed of execution. Besides, the user can run, pause and resume the program through the visual interface of HermesDebugger.

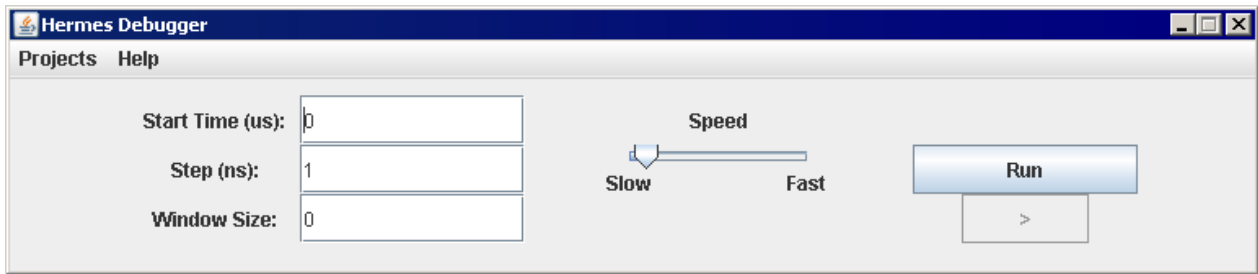


Fig. 2.19: Screenshot of HermesDebugger.

The “start time” parameter sets the simulation time on which the HermesDebugger should initiate presenting visual status of the NoC. This allows for example to jump the operating system booting process and debugging only communications related to the application under test. The “step” parameter defines the interval between two moments that have to be displayed by the scopes. This allows the user to configure if all events will be displayed by the scope or just an overview of the behavior of a simulation will be presented when the simulation is too long. The “window size” parameter sets the number of simulation cycles on which HermesDebugger should accumulate information by a given scope before considering the information old and deleting it. This feature is useful when the real time mode is too fast and a simple accumulation of data is meaningless for debugging purposes. The “window size” parameter affects only the scopes that work in sliding window mode as explained on Section 2.3.2, which are the EndToEndScope, HotSpotScope, InputScope, OutputScope and PowerScope. The “window size” parameter is measured in simulation cycles and if this parameter is set to zero it works in real time mode. The “speed” parameter sets the amount of time that the scopes should stay frozen for visualization purposes before updating the next event on the scope.

For example, if a program execution is initiated with parameters “step time” equals to 100 ns and “speed” equals to 100 ms, that means, the scopes will first display the results for simulation time equals to 0 ns and keep this result in the screen for at least 100 ms; then, the monitors will display the results for simulation time equals to 100 ns and keep

this result in the screen for at least another 100 ms; then, again the time is increased by 100 ns and so on. This kind of control is required for not losing the concept of time during the visual debugging of the NoC.

The following Sections present detailed information of which HERMES NoC signals are important for each scope of the NoCScope.

2.4.1 BufferScope

The BufferScope measures the number of occupied positions in each port of each router. This scope demands a very low computation power, since each buffer is controlled by two pointers: *last* and *first*. *Last* points to the last occupied position of the buffer, while *first* points to the first occupied position. Thus, in order to find the number of flits stored in the buffer, one simple subtraction is enough ($buffer_size = last - first$).

Since the buffer used in the HERMES NoC is a circular buffer, when *last* reaches the size of the buffer, the next position will be again “zero”. This is employed to simplify the hardware of the buffer. Therefore, if the result of the subtraction is negative, the value of buffer size must be corrected by adding the buffer depth to it.

2.4.2 InputScope

The InputScope counts the number of flits received by each input port of each router. In order to know if a packet arrived in a specific input port, the program just needs to analyze the *rx* signal. If the bit associated to the chosen port is asserted high, then the counter of this port should be incremented. However, the *rx* signal can remain set by many cycles for the same flit. Thus, only positive edge transitions are considered in this scope.

2.4.3 OutputScope

The OutputScope counts the number of flits sent by each output port of each router. In order to know if a packet left a specific output port, the program just needs to analyze the *tx* signal. If the bit associated to the chosen port is asserted high, then the counter of this port should be incremented. However, the *tx* signal can remain set by many cycles for the same flit. Thus, only positive edge transitions are considered in this scope.

2.4.4 EndToEndScope

The EndToEndScope only requires information about the source and the destination routers of a packet. Then, this scope is based on detecting whenever a packet enters or exits the network, and storing the information contained in its header.

When a packet enters the network, the local port of the router receives the first flit of packet, stores it in the input buffer and requests arbitration and routing. When the *ack_h* signal of the local port is activated, the packet has successfully entered the network and the *localx*, *localy*, *targetx* and *targety* signals are stored in a table as a new packet currently

inside the network. When the packet arrives to its destination ($localx=targetx$ and $localy=targety$) the connection entry is removed of the table.

2.4.5 PointToPointScope

Among all scopes, the PointToPointScope is the one that demands more computation power, since the HermesDebugger has to store all information about all active router connections in a table. Every router and port passed by a packet is stored in this table, so the HermesDebugger can paint the entire path of the packet in the scope. When a packet is passing by a local port of a router, as explained in Section 2.4.4, this router is either the input or output access point of a packet in the network. If none of the previous cases are true and a packet is forwarded by a router, it means the current router is an intermediate hop. In all cases, after routing is done (ack_h is activated) and an input port (informed by signal sel) is connected to an output port (indicated by using sel as an index of the mux_in array), this information is stored in a table and used to update the PointToPointScope.

2.4.6 HotSpotScope

The HotSpotScope quantifies the output ports of the NoC routers that are trying to send packets ahead to a neighbor router, but are being blocked due to congestion. The calculation is based on a simple analysis of two signals. If the signal rx of a port is set, while the acknowledgment signal (ack_rx for handshake and $credit_o$ for credit based) is not, that means some port tried to communicate with this one, but the connection was denied due to congestion problems. Thus, the hotspot counter should be incremented.

2.4.7 PowerScope

The PowerScope estimates the power consumption of each port by comparing the current flit with the previous flit and, then, counting the number of transitions occurred in each bit. Therefore, the implementation consists only in storing the two most recent flits received by the $data_in$ signal, applying the “exclusive or” operation between both values, and counting the number of bits ‘1’ in the resulting vector.

2.5 Summary

This Chapter presented an introduction on modeling NoC infrastructures using actor-orientation (Sections 2.1 and 2.2.2). The HERMES RTL NoC (Section 2.2.3) and a visual debugging system for HERMES (Section 2.4) were presented. Three NoC models (RENATO – Section 2.2.4, JOSELITO – Section 2.2.5 and BOÇA – Section 2.2.6) based on HERMES were implemented and compared (Section 2.2.7). A monitoring and debugging system for RENATO was created (Section 2.3).

All these issues are important to create communication infrastructures for MPSoCs, because designers should be able to analyze different alternatives for the on-chip

interconnect, aiming to satisfy the particular requirements of performance, area, cost and power consumption for a given application domain. Thus, it is important to dispose models that allow the simulation of the NoC in different levels of abstraction, considering different architectures and traffic conditions. Fast and abstract platform models should be used early on the design flow in order to perform rough evaluations and to rule out poorly performing platforms. Accurate models should be used later for fine-tuning platform parameters and choosing the best mapping.

By following the presented approach, a designer can quickly change buffer schemes, routing and arbitration algorithms, and even the network topology, then simulate the model in Ptolemy II and obtain figures for performance, area and power consumption which are reasonably accurate, all within minutes. Obviously the figures can't be as detailed and accurate as the ones obtained by RTL simulation, but if the loss of accuracy is acceptable for a system-level analysis it can be compensated by the ease of change of the model and the simplicity on displaying the results.

On this work an interaction-based analysis methodology was used to abstract the functionality of a RTL NoC implementation. The flexibility of this methodology is such, that different configurations of the same NoC (or even different NoCs with slightly different behavior) can be quickly modeled simply by annotating the timing information for each individual component. Based on the interactions, formalized using UML sequence diagrams, actor-oriented models of the NoCs were built using Ptolemy II. The proposed approach is not restricted to the HERMES NoC or to the Ptolemy II framework. It could be applied to each and every NoC (as long as its interactions can be captured as shown in Section 2.2.4) and the corresponding actor-oriented model could be built on any simulator supporting multiple models of computation.

To validate the accuracy of the proposed approach, the performance results of the actor-oriented model RENATO were compared with the results of the cycle-accurate simulation of HERMES, showing errors in the order of 10% for long-lasting traffics, which are acceptable for design space exploration. With the payload abstraction technique used by JOSELITO, the execution of the modeled application was in average 2.3 times faster than RENATO. Additionally, JOSELITO was compared to the cycle-accurate simulation of HERMES and presented worst case simulation latency and throughput error results of 5.26% and 0.1%.

Besides the performance analysis, the actor-oriented models can provide three levels of debugging and observability, including graphical information of buffer occupation, channel load, network congestion and power consumption figures. This extended observability of the NoC can be used for experimenting different arbitration and routing algorithms, testing new task mapping or task migration algorithms, creating new network topologies or just changing the size of it, testing new buffering strategies or just setting new buffer sizes, analyzing different flow control schemes, finding a good trade-off between the number of packets and the packet length used by the network interfaces of the modules connected to the NoC, and changing the flit size to see the new latency results provided by the tool.

By attaching the NoCScope to the HERMES RTL NoC through the HermesDebugger tool, an extended observability of the RTL NoC can be achieved, allowing the quick identification of similar problems that were only visible in the NoC model without requiring the analysis of waveforms. This was only possible by exporting a list of events of the HERMES NoC simulated by ModelSim, and translating these events in a format accepted by the NoCScope.

3 Application and MPSoC Modeling[†]

The previous Chapter has presented different NoC models that can have several parameters configured to achieve a flexible communication infrastructure. However, a NoC (and consequently, an MPSoC) can only be configured after an application or a group of target applications are known. Therefore, this Chapter will first focus on how to model applications for later mapping these applications to an MPSoC.

3.1 Application Modeling

Building an application is a maturation process. First it is necessary to know which inputs should be given to the application and which kind of outputs should be resulted. Then, the application should be broken into blocks, and these blocks could be possibly implemented by a designer or a group of designers. The implementation process can take months. After each block is done and properly tested, these blocks are connected among them and tested again. If problems are found, designers have to go back to the implementation phase and fix the problems. This loop can also take months. Finally, after all blocks are connected and working properly, the system is evaluated to see if it meets the expected speed, area and power consumption requirements.

From this simplified summary about building an application, one important conclusion can be extracted: if the final system does not meet the expected requirements, the company that pays the designers have lost the time to market to release a new product and more time will be required for the designers to finish this product. The competitor will probably not loose this opportunity, selling its products to more customers.

Modeling the application early at the design process may be the key point to avoid such a failure. An application model can simplify the implementation phase by abstracting the computation of one or more blocks of the application. At one hand, it is possible to see early in the design process all blocks interconnected together and fix/avoid possible problems related to this interconnection. This would reduce the number of bugs/tests required to complete the application and provide an initial feedback of the interconnection's performance, which could in turn provide initial figures regarding speed and power consumption. On the other hand, the lack of real computation could be producing results that could mislead the designers to wrong conclusions. The goal is to try to quickly produce an amount of data that should be exchanged among blocks

[†] Major parts of this Chapter were presented at ReCoSoC [108] and IJERTCS [112].

which is expected on the final application. Three different ways for providing this data will be presented in this Chapter: (i) each block is abstracted to static data read from a text file in periods of time decided at design time, as presented on Section 3.1.1; (ii) each block is abstracted to an Instruction-Set Simulator (ISS), providing some sort of computation and data decided at runtime, as presented on Section 3.1.2; (iii) each block is abstracted to a lifeline of a UML sequence diagram, providing just the expected computation time and the expected amount of communication decided at runtime, as presented on Section 3.1.3.

3.1.1 Synthetic Traffic

Synthetic traffic is a data stream created to mimic the behavior of an application. As presented by Tedesco et al. [27], three different methods can be identified on the literature to model traffic with the goal to characterize complex applications. *“The first one assumes that sources continually send data at a constant rate to the network. This is the most commonly used method. The second method employs probabilistic functions to model the traffic behavior for typical applications, as audio and video streams. The accuracy of this method is better, at the extra cost of modeling complexity and simulation time. The third method employs traffic traces to evaluate network performance. Even with small traces, simulation time can be prohibitive. The advantage is accuracy, superior to the previous models. Even if a given application is correctly modeled, other flows interfere on how the application traffic behaves within the network [27]”*.

In this work only the first two methods were explored on the initial evaluation of the NoC models RENATO and JOSELITO, which were presented on Section 2.2.4 and 2.2.5. On Section 2.2.7 a synthetic traffic was also used to create the autonomous vehicle application, but no standard probabilistic model fit well to the application. Therefore, a specific synthetic traffic was created to model the application, but the interactions between tasks of the application were severely compromised. The main problem was that an answer message did not wait for a request message to be transmitted. This happens because it is the nature of the synthetic traffic to act according to a predefined traffic file that contains which messages have to be sent at specific time points. And if the NoC experiences extra delays (e.g. different traffic flows created a temporarily congestion point on the NoC), then a supposed answer message can be sent before the receipt of the supposed request message. For this reason, synthetic traffic is no longer used on this work from this point on.

3.1.2 Instruction Set Simulator

Another alternative that can be used in combination with synthetic traffic is an Instruction Set Simulator (ISS). The main advantage of an ISS is that both the implementation of communication and computation of an application can be modeled. The main disadvantage is that the simulation of the model will require longer time than synthetic traffic only.

Tab. 3.1 summarizes the most important MPSoCs that use ISSs to model or debug applications. As presented in Tab. 3.1, all works use SystemC as simulation engine and memory mapped techniques to communicate with other processors, except the work proposed here that uses the Ptolemy II simulation engine and the message passing technique to communicate with other processors.

Tab. 3.1: MPSoCs that have tools for debugging embedded software.

Work ID	Simulation engine	Processor	Communication	Data exchange	ISS	OS
MPARM	SystemC	ARM	Bus (Amba)	Memory	SWARM	uClinux
STARSoC	SystemC	OpenRisc 1200	Bus (Wishbone)	Memory	OR1Ksim	No
HVP	SystemC	Several	Bus (several)	Memory	Several	Yes
SoClib	SystemC	Several	Bus / NoC (several)	Memory	GDB	several
Proposed	Ptolemy II	Plasma (MIPS)	NoC (Hermes)	Message	MARS	No

MPARM [28] uses ARM processors connected through AMBA bus to compose the MPSoC. Multiprocessor applications are debugged with the SWARM ISS, which is developed in C++ and was wrapped to communicate with the MPSoC simulated in SystemC. The platform allows booting multiple parallel uClinux kernels on independent processors.

STARSoC [29] uses OpenRisc1200 processors connected through Wishbone bus. Debugging is implemented with the OR1Ksim ISS, which is implemented in C language. The OR1Ksim also allows to be remote operated using GDB. Operating System is not yet supported.

HVP [30] supports several processors and therefore several ISSs. The work presented MPSoCs that contain ARM9 processors using ARM's ISS and in-house VLIW and RISC processors debugged by the LisaTek ISS. The ARM processors execute a lightweight operating system (name was not disclosed). The communication among processors was reported to be AMBA among ARM processors and SimpleBus among the in-house processors used.

SoClib [31] is a project developed jointly by 11 laboratories and 6 industrial companies. It contains simulation models for processor cores, interconnect and bus controllers, embedded and external memory controllers, or peripheral and I/O controllers. The MPSoC accepts the following processor cores: MIPS-32, PowerPC-405, Sparc-V8, Microblaze, Nios-II, ST-231, ARM-7tdmi and ARM-966. The GDB client/server protocol has been implemented to interface with these processors. The following operating systems are supported: DNA/OS, MutekH, NetBSD, eCos and RTEMS. Several bus and NoCs with different topologies wrapped with the VCI communication standard were ported and presented at [32].

The proposed work is based on a MIPS-like processor, implemented in hardware by the Plasma processor available for free at Opencores [33] and implemented by MARS [34] when simulating the processor as an ISS. While all previous works use SystemC as

simulation environment, this work uses the Ptolemy II [35] presented on Section 2.1.2. Inside the Ptolemy II environment, the MARS ISS was connected to the RENATO NoC presented on Section 2.2.4, thus creating an MPSoC model capable of considering the computation of the application and the timing delays of the NoC. This work also differs from the others because it exchanges data between processors by using the native protocol of the NoC, therefore no extra translation is needed before sending and receiving packets.

MARS is an Instruction Set Simulator (ISS) developed by Peter Sanderson and Kenneth Vollmar, from the Missouri State University [36]. MARS assembles and simulates programs written in the MIPS assembly language. It contains an Interactive Development Environment (IDE) presented on Fig. 3.1 for visually debugging MIPS programs, but it can also run from command line for generating batch scripts. MARS is open-source software available on [37] and is written entirely in Java. MARS can simulate 155 basic instructions from the MIPS-32 instruction set, as well as about 370 pseudo-instructions or instruction variations, 17 *syscall* functions for console and file I/O and 21 *syscalls* for other uses. It supports seven different addressing modes as well: *label*, *immed*, *label+immed*, *(\$reg)*, *label(\$reg)*, *immed(\$reg)*, and *label+immed(\$reg)*, where *immed* is an integer up to 32 bits. There is support for disabling pseudo-instructions, extended instruction formats and/or memory addressing modes [37].

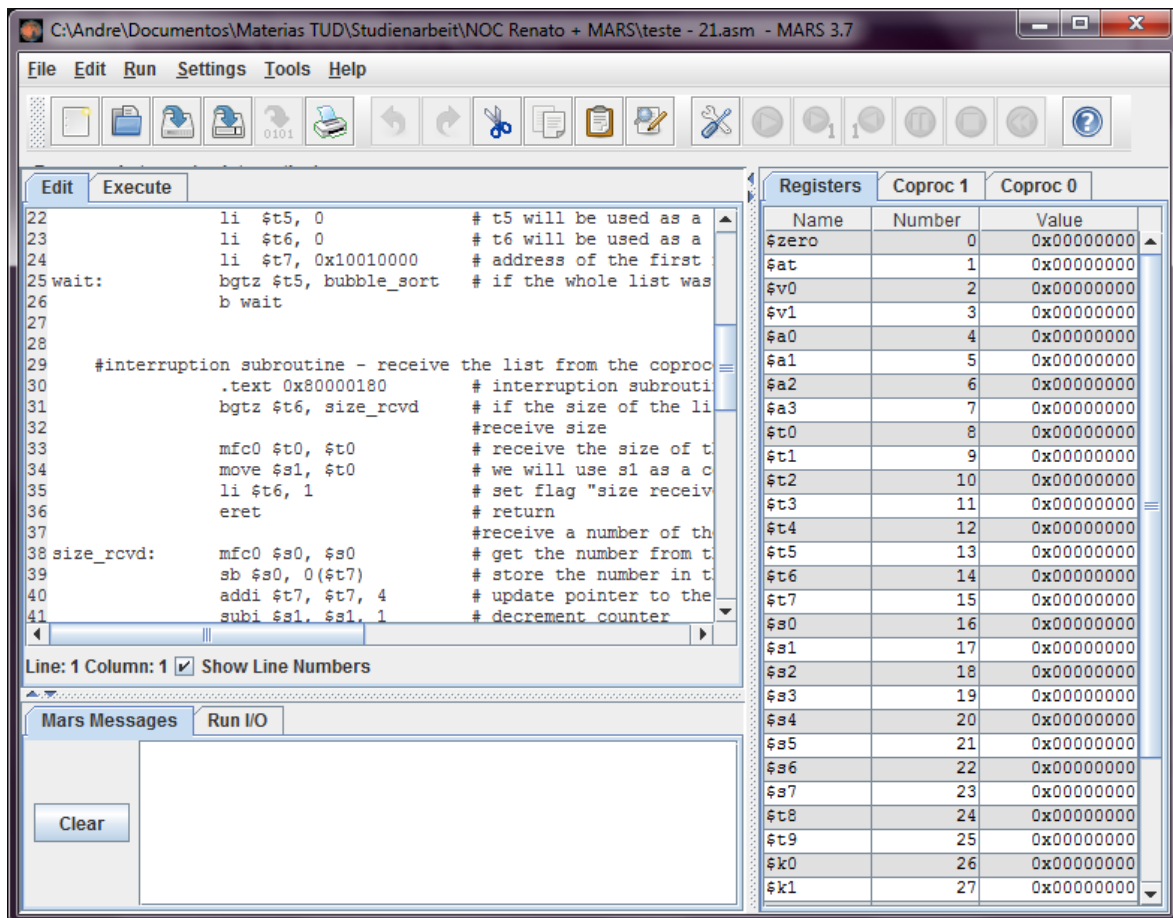


Fig. 3.1: MARS 3.7 IDE.

The following Sections present how MARS was connected to the RENATO NoC to allow the creation of an MPSoC model. Fig. 3.2 shows a block diagram of the system that will be used in the next Sections to guide the explanation of each important item.

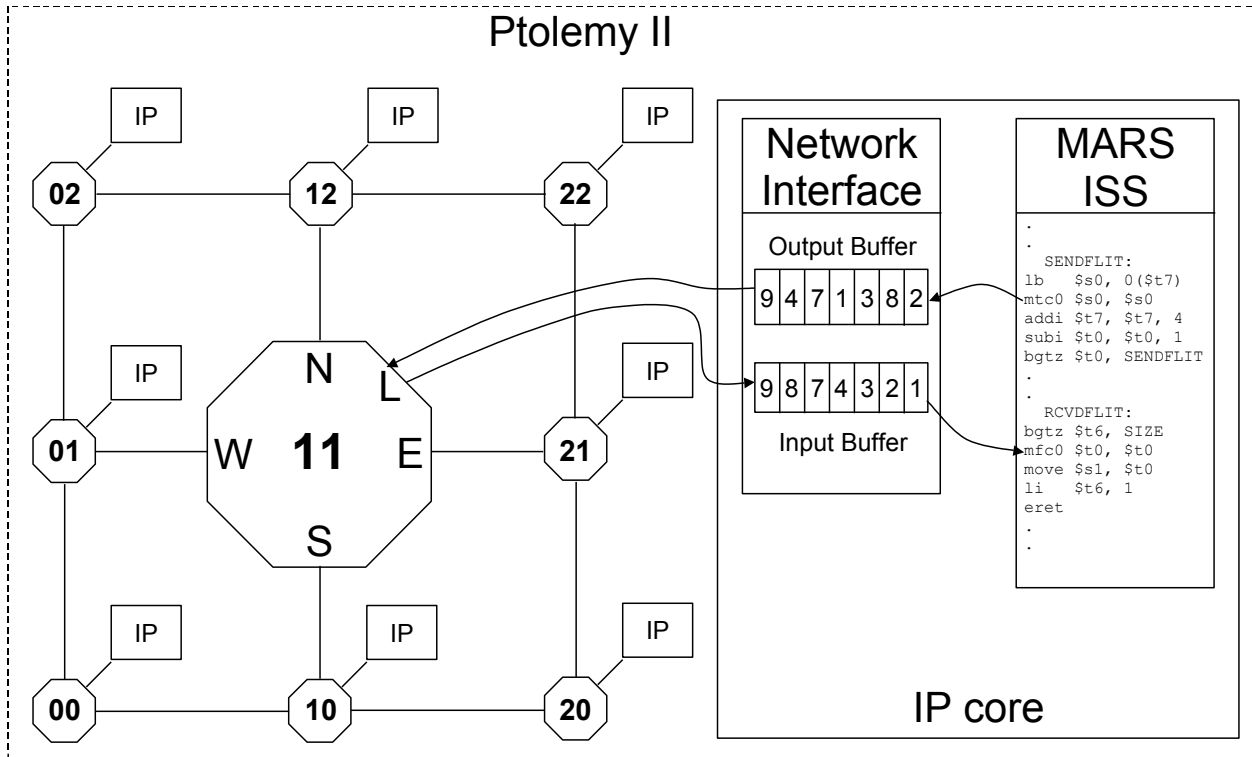


Fig. 3.2: Block diagram of the proposed MPSoC model connected to the MARS ISS.

3.1.2.1 Processor to NI

In the current version of this work, each processor executes the MIPS assembly code of one task of the application (mono-task). Communication between tasks happens by exchanging packets. In order to send a packet to another task, the header of the packet and the packet data need to be first stored in the data memory of the processor by the task. The header of the packet is composed by the address of the target router where the processor is connected and the number of data flits this packet contains. After that, the send packet subroutine is called.

The send packet subroutine first reads the size flit of the packet stored in the memory to a register and reads to another register the output buffer size available in the NI. If there is enough space available in the NI to store the packet, the subroutine proceeds sending the packet flit by flit to the NI. The process of “reading” a flit from the NI uses the instruction “move from coprocessor 0” (*mfc0*), while the process of “sending” a flit to the NI uses the instruction “move to coprocessor 0” (*mtc0*). Thus, from the point of view of the processor, coprocessor 0 is now the NI.

3.1.2.2 NI to NoC

With the packet stored in the NI output buffer, the NI sends the packet flit by flit to the input local port of the router where this NI is connected. This happens following the

flow control protocol in use by the NoC and using the timing delays set on the NoC model being executed by Ptolemy.

3.1.2.3 NoC to NI

When packets are being received from the NoC into the NI, a different buffer (input buffer) is used, thus allowing parallel sending and receiving of packets. The receiving of packets also occur following the flow control in use by the NoC and using the timing delays set on the NoC model.

3.1.2.4 NI to processor

As soon as the flits of the packet arrive in the input buffer of the NI, the NI launches a specific interruption to the processor meaning that a new packet has arrived. The MARS ISS, which was executing its task, saves its context and receives the interruption in the form of a Java exception. The standard routine for handling exceptions is called. By the ID of the specific exception, the exact exception is found out to be the “new message from network exception”. The specific subroutine of this exception is launched. This subroutine mainly reads the complete packet from the NI using the “move from coprocessor 0” (*mfc0*) instruction to read each flit of the packet. After the complete packet was read from the NI and stored in the processor’s memory, the processor’s context is restored and it can now continues with its execution possibly using the data that was received.

3.1.2.5 Synchronization

The straightforward solution in Java to connect more than one MARS ISS to the NoC is to create a new MARS instance object for every new MARS instantiated in the NoC. However, this alternative failed due to the fact that MARS has been programmed using several static classes, attributes and methods. All of its main resources, such as the memory and the register bank, are declared as static. Therefore, if one tries to run more than one instance of MARS concurrently inside a single Java Virtual Machine (JVM), all the running instances will share the same resources, which will lead to unexpected behavior.

One possible workaround for this problem is to run each MARS instance in a different JVM. Java does not directly share memory between multiple VMs, so by running each MARS in a different JVM, one is safely isolating each instance of MARS. One problem with this approach is that the exchange of messages between different JVMs is only possible by using APIs such as Java Remote Method Invocation (RMI) and sockets, which would greatly increase the complexity of the system.

Another solution would be to reprogram MARS to remove the problematic static attributes and make them unique for each instance. However, this solution was also not optimal, considering the large number of static members declared in MARS and that every new future version of MARS would also require these modifications.

A better solution is to instantiate isolated ClassLoaders, one for each instance of MARS to be loaded. This works because a static element in Java is unique only in the context of a ClassLoader, therefore the static elements will not interfere with the other instances of MARS called by other ClassLoaders. By using this approach, the task of exchanging messages between the MARS instance and its corresponding NI also becomes trivial, and can be done simply by injecting a NI object when instantiating MARS.

A side effect of this solution is that each MARS instance and the NoC are considered as different threads by Java, and this would require extra algorithms based on *wait* and *notify* directives to maintain the time constraints followed by the NoC. As this would slow down the simulation of the system by forcing a specific scheduling of heavy threads (Ptolemy II and MARS) to the JVM, which would require many more context saving and restoring, the NoC is working according to one “clock domain” while each MARS instance works according to a different “clock domain”. This issue implies no error to the system because even if a MARS instance is executing faster than the NoC, the NoC serializes packet transfers to its own “clock domain”. It is being used the term “clock domain” here quoted because we are borrowing a hardware concept and using in a software environment, which describes more clearly the timing issues that different threads face.

Fig. 3.3 presents a printout of the most important events occurred during the transfer of a packet composed by 2 header flits and 10 payload flits from MARS #1 to MARS #2. MARS #1 is connected to router 00 as illustrated in Fig. 3.3 and MARS #2 is connected to router 21. No extra traffic is currently occupying the NoC. All the following comments presented in this paragraph refer to Fig. 3.3. Between times 3002 and 3086 MARS #1 sends the packet to the NI connected to it (NI #1), exactly as explained in Section 3.1.2.1. Eleven of the twelve flits of the packet were sent in the first 2 simulation cycles, and the last flit of the packet at time 3086. This strange behavior implies the following results: (1) MARS #1 thread was executed two times concurrently to Ptolemy thread, between times 3002-3003 and 3086; (2) MARS thread can be faster enough to execute at least 11 *mtc0* instructions in a row during 2 simulation cycles of Ptolemy; (3) MARS thread was not called again during 83 simulation cycles (3086-3003). Between times 3087 and 3109 each flit of the packet was sent constantly every 2 simulation cycles from NI #1 to the NoC, exactly as explained in Section 3.1.2.2. This behavior is equal to the real HERMES NoC that needs 2 clock cycles to transfer a flit using handshake flow control. Between time 3112 and 3156 all the flits from the packet were delivered from the NoC to NI #2 as explained in Section 3.1.2.3. However, due to some technical difficulties in the current version, it was not possible to deliver each flit every 2 simulation cycles, but 4 simulation cycles in this case. At time 3120 it is possible to see that NI #2 delivered the first payload flit immediately to MARS #2. Between times 3166 and 3181 the rest of the payload flits were delivered to MARS #2 as described in Section 3.1.2.4. Here again it is possible to see that the data transfer did not follow a constant pattern, similar to one the occurred between times 3002 and 3086. This unpredictable behavior is a side effect of running multiple threads with no proper synchronization.

3002	MARS #1	sending target	flit (21)	to NI #1
3002	MARS #1	sending size	flit (10)	to NI #1
3002	MARS #1	sending payload	flit #0 (9)	to NI #1
3003	MARS #1	sending payload	flit #1 (9)	to NI #1
3003	MARS #1	sending payload	flit #2 (4)	to NI #1
3003	MARS #1	sending payload	flit #3 (7)	to NI #1
3003	MARS #1	sending payload	flit #4 (1)	to NI #1
3003	MARS #1	sending payload	flit #5 (3)	to NI #1
3003	MARS #1	sending payload	flit #6 (8)	to NI #1
3003	MARS #1	sending payload	flit #7 (2)	to NI #1
3003	MARS #1	sending payload	flit #8 (6)	to NI #1
3086	MARS #1	sending payload	flit #9 (5)	to NI #1
3087	NI #1	sending target	flit (21)	to NoC
3089	NI #1	sending size	flit (10)	to NoC
3091	NI #1	sending payload	flit #0 (9)	to NoC
3093	NI #1	sending payload	flit #1 (9)	to NoC
3095	NI #1	sending payload	flit #2 (4)	to NoC
3097	NI #1	sending payload	flit #3 (7)	to NoC
3099	NI #1	sending payload	flit #4 (1)	to NoC
3101	NI #1	sending payload	flit #5 (3)	to NoC
3103	NI #1	sending payload	flit #6 (8)	to NoC
3105	NI #1	sending payload	flit #7 (2)	to NoC
3107	NI #1	sending payload	flit #8 (6)	to NoC
3109	NI #1	sending payload	flit #9 (5)	to NoC
3112	NoC	sending target	flit (21)	to NI #2
3116	NoC	sending size	flit (10)	to NI #2
3120	NoC	sending payload	flit #0 (9)	to NI #2
3120	NI #2	sending payload	flit #0 (9)	to MARS #2
3124	NoC	sending payload	flit #1 (9)	to NI #2
3128	NoC	sending payload	flit #2 (4)	to NI #2
3132	NoC	sending payload	flit #3 (7)	to NI #2
3136	NoC	sending payload	flit #4 (1)	to NI #2
3140	NoC	sending payload	flit #5 (3)	to NI #2
3144	NoC	sending payload	flit #6 (8)	to NI #2
3148	NoC	sending payload	flit #7 (2)	to NI #2
3152	NoC	sending payload	flit #8 (6)	to NI #2
3156	NoC	sending payload	flit #9 (5)	to NI #2
3166	NI #2	sending payload	flit #1 (9)	to MARS #2
3170	NI #2	sending payload	flit #2 (4)	to MARS #2
3172	NI #2	sending payload	flit #3 (7)	to MARS #2
3174	NI #2	sending payload	flit #4 (1)	to MARS #2
3175	NI #2	sending payload	flit #5 (3)	to MARS #2
3177	NI #2	sending payload	flit #6 (8)	to MARS #2
3178	NI #2	sending payload	flit #7 (2)	to MARS #2
3180	NI #2	sending payload	flit #8 (6)	to MARS #2
3181	NI #2	sending payload	flit #9 (5)	to MARS #2

Fig. 3.3: Timing delays of the most important events during the transfer of a packet between two processors.

3.1.3 UML Sequence Diagrams

A third alternative used on this work to model applications is through the use of UML sequence diagrams. UML is a short for Unified Modeling Language and it is a language for specifying, constructing, visualizing and documenting the artifacts of a software-intensive system [38]. This language fuses the concepts of Booch, OMT and OOSE, which were created by Grady Booch, Jim Rumbaugh and Ivar Jacobson, respectively. The diagrams supported by UML are: class, component, composite structure, deployment, object, package, profile, activity, communication, interaction overview, sequence, state, timing and use case.

The sequence diagram is of special interest for this work due to its simplicity to model the exchange of messages between tasks. With this motivation in mind, Indrusiak et al. [39][40][19] have benefited from the flexibility of the Ptolemy II framework presented on Section 2.1.2 to create an executable application model based on UML sequence

diagrams for describing the communication patterns of the application and encapsulating the diagrams inside actors. A key element of the work is the implementation of two different directors for controlling the execution of sequence diagrams, total order and partial order, based on SDTODirector and SDPODirector respectively [40]. The total order director maintains the order of the messages of a sequence diagram whereas the partial order director maintains the order separately on each lifeline. Both of the directors create a precedence graph of all messages of the sequence diagram. Fig. 3.4 depicts the difference between total and partial order precedence graphs. Total order sequence diagram can be illustrated using a directed graph, where the precedence of messages M1-M4 is the same than their order in the sequence diagram. Partial order is also a directed graph, but now it is possible to see that the order of messages M1 and M2 does not matter, because the order of messages is enforced separately on each lifeline and M1 and M2 do not have events on the same lifeline.

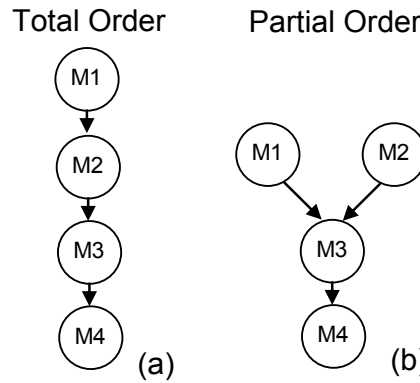


Fig. 3.4: Difference between Total Order and Partial Order precedence graphs [112].

Fig. 3.5 presents a simplified block diagram of the different components used to model MPSoCs (for a formal explanation please refer to [112]). An application can be modeled by connecting ‘application actors’ to ‘sequencing actors’. Each ‘application actor’ can be as complicated as a processor executing several tasks or as simple as a simple logic creating new events. The important is that every communication between an application actor ‘a’ to an application actor ‘b’ has to occur by sending ‘tokens’ through a ‘sequencing actor’. A ‘token’ can be any kind of value (e.g. integer, float, string, array, ...). The ‘sequencing actor’ uses a UML sequence diagram to define which ‘application actors’ should communicate with which others and on which sequence. Application actors are called ‘lifelines’ inside the UML sequence diagram, the communication between them is specified by using ‘messages’. Every ‘message’ has two main attributes: size (amount of bytes to be sent to target task) and computation time (amount of time that the task should wait before sending this message after the previous message was received). Special ‘directors’ were created to specifically control the execution semantics of the UML sequence diagram, as explained on the previous paragraph. The ‘mapper’ is responsible to map ‘application actors’ to ‘IP cores’ on the platform. The platform is composed by ‘IP cores’ interconnected by a NoC. The NoC is composed by ‘routers’ interconnected by ‘channels’. The IP cores receive tokens created by the application

actors on the ‘producer’ component. The producer sends the token to the NoC, where the token experience the delay of the NoC, and after passing through possible congestions arrive to the target IP core. The ‘consumer’ component of the IP core target is responsible to forward this token to the target ‘application actor’.

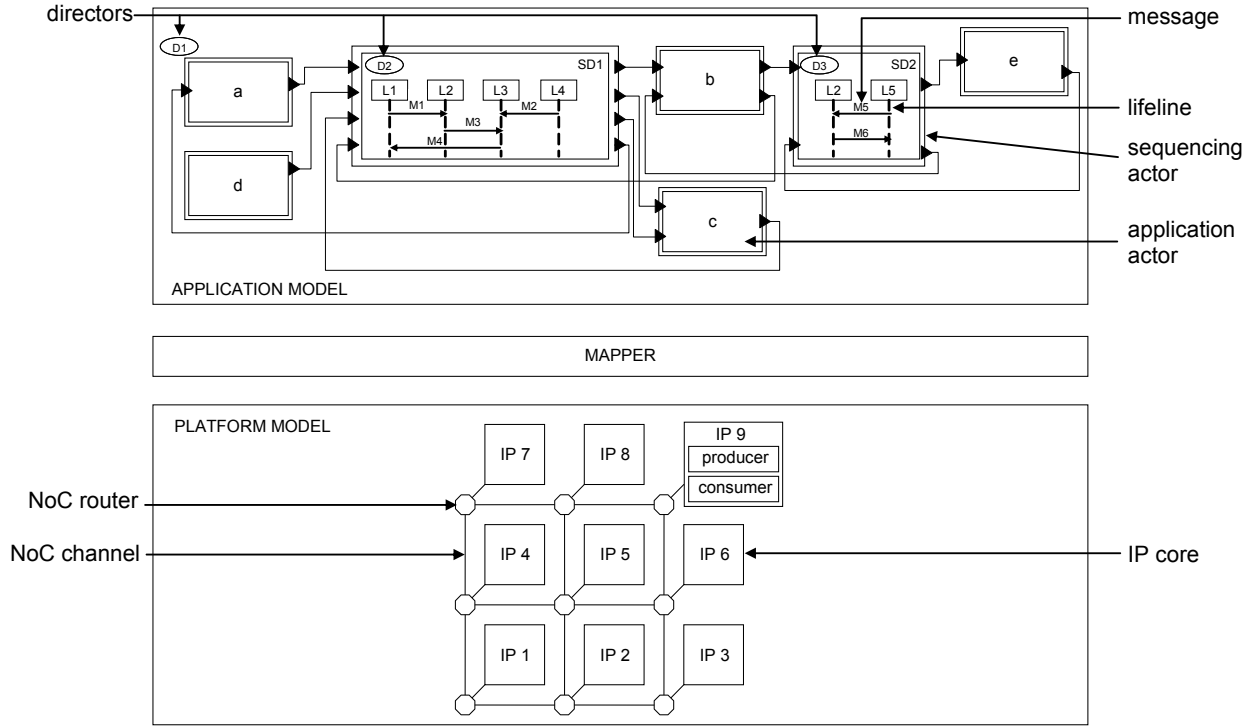


Fig. 3.5: Joint validation of application and platform [112].

The first lifeline to send a message on a sequence diagram (e.g. L1 of SD1 and L5 of SD2 on Fig. 3.5) has to refer to an ‘initiator actor’, which is an application actor that create tokens. On the other hand, a ‘passive actor’ is an application actor that reacts or initiates a communication after receiving a message from an initiator actor [41].

3.2 Heterogeneous MPSoC

Until now the MPSoC model presented on this work was homogeneous in nature, that is, all IP cores connected to the MPSoC model were identical, and all application tasks were allowed to be mapped on any IP core. If the MPSoC model is extended to a heterogeneous system, different types of IP cores can co-exist on the MPSoC, and thus provide IP cores with different characteristics. This opens the possibility to use the most efficient IP core to a given task, or even to implement the same task for multiple IP cores and let the system to automatically trade flexibility, power consumption and performance at runtime.

The aforementioned behavior of the IP cores connected to the heterogeneous MPSoC model has a tight similarity with a spectrum of technologies illustrated on Fig. 3.6. GPPs are flexible for accepting software tasks and for handling concurrent tasks (in case that the GPP runs at least a micro-kernel that supports multi-tasking). On the other side of the spectrum, an ASIC is only able to perform a unique specific task, but this task is

executed with highest performance and is an obvious choice when this task is executed all the time on a system.

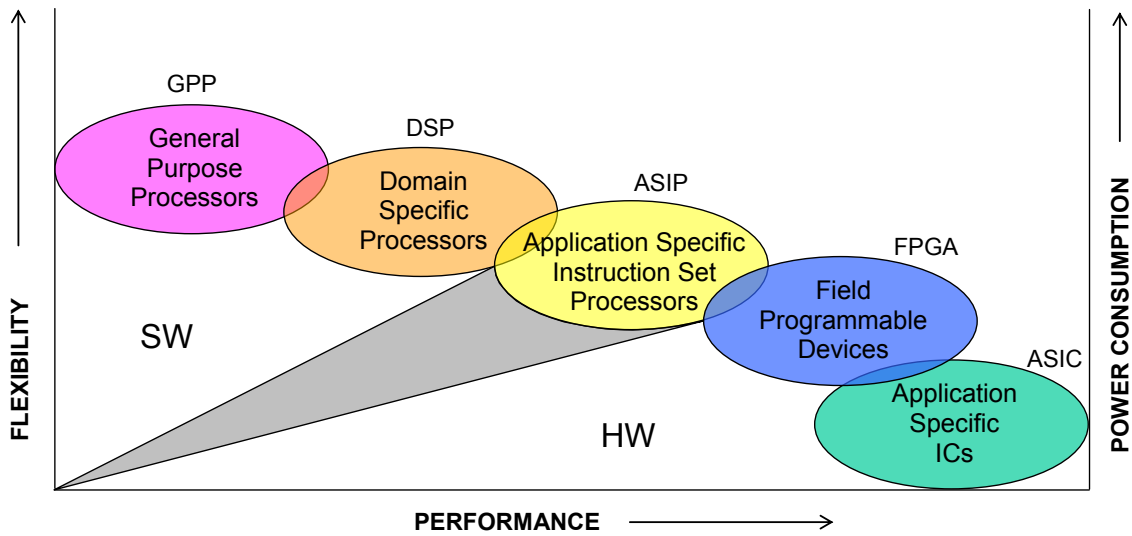


Fig. 3.6: Different technologies and their general characteristics. Adapted from [42].

This comparison between different technologies and their characteristics (flexibility, performance and power consumption) is a valuable source of insight for associating tasks to IP cores. Such association is especially important for the work proposed here, because the system relies completely on the know-how of the designer for setting parameters related to the performance of a task with regard to each possible IP core that can implement it. For this purpose, a type system and application constraints are introduced to the MPSoC model on the next Sections, and based on these enhancements, new mapping assignments, multi-tasking management, and task migration algorithms are presented.

3.2.1 Type System

In real world scenario, some IP cores are more optimized for particular type of applications or tasks. For example, Digital Signal Processors (DSPs) are far more efficient for extensive numerical real-time computations than General Purpose Processors (GPPs).

In order to include abstract solution for such a heterogeneous MPSoC, a type system was introduced on the platform model. For instance, IP cores can be categorized according to five types, where each type represents an instance of a particular IP core. For example: type 0 could represent General Purpose Processor (GPP) and type 1 could represent a Digital Signal Processor (DSP). Fig. 3.7 shows the parameters of an IP core named 'prod30' with network address 30 and its 'Type' being specified as '1'.

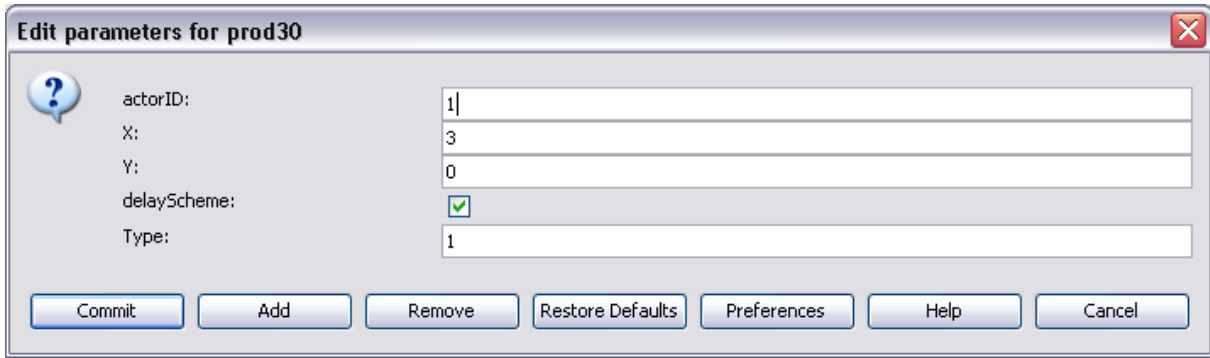


Fig. 3.7: Parameters of an IP core.

3.2.2 Task Parameters

After the introduction of the type system, an application task has to be bounded to at least one type of IP core. In case of a heterogeneous system, an application task can have type compatibility to more than one IP core, while for a homogeneous system, all application tasks support only one type of IP core. Type compatibility in this context means that the system contains the object code of the task for a certain IP core. To improve the characterization of a task for a given IP core and improve the quality of the mapping algorithms, the parameters computation time, affinity, memory footprint and utilization should be defined for every task. **Computation time** is how many simulation cycles are required for the task to execute its function. Sometimes the computation time may depend on the value used as input for the task. In such cases it is common in real time systems to define it as the worst case computation time to guarantee that the task will always be able to execute without missing the deadline. Some other times the computation time may depend on which task sent a requesting message for the current task. In such cases the application can be more accurately modeled if this distinction is considered. Therefore, this work binds the computation time not to a task, but to a message sent by one task. Fig. 3.8 presents this situation where message 'm3' and 'm5' belong to task 'IMP', that have different computation times (1,000 and 3,000, respectively) because they react to different messages ('m2' and 'm4', respectively) sent by different tasks ('MC' and 'FB1', respectively).

Affinity is measured in percentage and it is a multiplicative factor to increase or reduce the computation time depending on which IP core the task is mapped. Every task must have its computation time defined in relation to the IP core over which the task has greater affinity. So, the computation time of a task t mapped on an IP core k (CT_{tk}) can be calculated by

$$CT_{tk} = \frac{\sum_{i=0}^m CT_i}{Af_{tk}} \quad \text{Eq. 3.1}$$

where m is the total amount of messages generated by task t , CT_i is the computation time of each message generated by t when mapped on an IP core which has 100% affinity and Af_{tk} is the affinity of the task t to the IP core k . **Memory footprint** is the size of the task measured in bytes. This size represents the object code and the context of the

task, which represent the volume of data that should be transferred through the NoC in case of migrating the task to a different IP core. **Utilization** is the total amount of computation time that a task demands from an IP core, and can be calculated by

$$U_t = \frac{CT_{tk}}{P_t} \quad \text{Eq. 3.2}$$

where U_t is the utilization of a task t , CT_{tk} is the computation time of a task t when mapped on the IP core under consideration k calculated by Eq. 3.1 and P_t is the period of the task t .

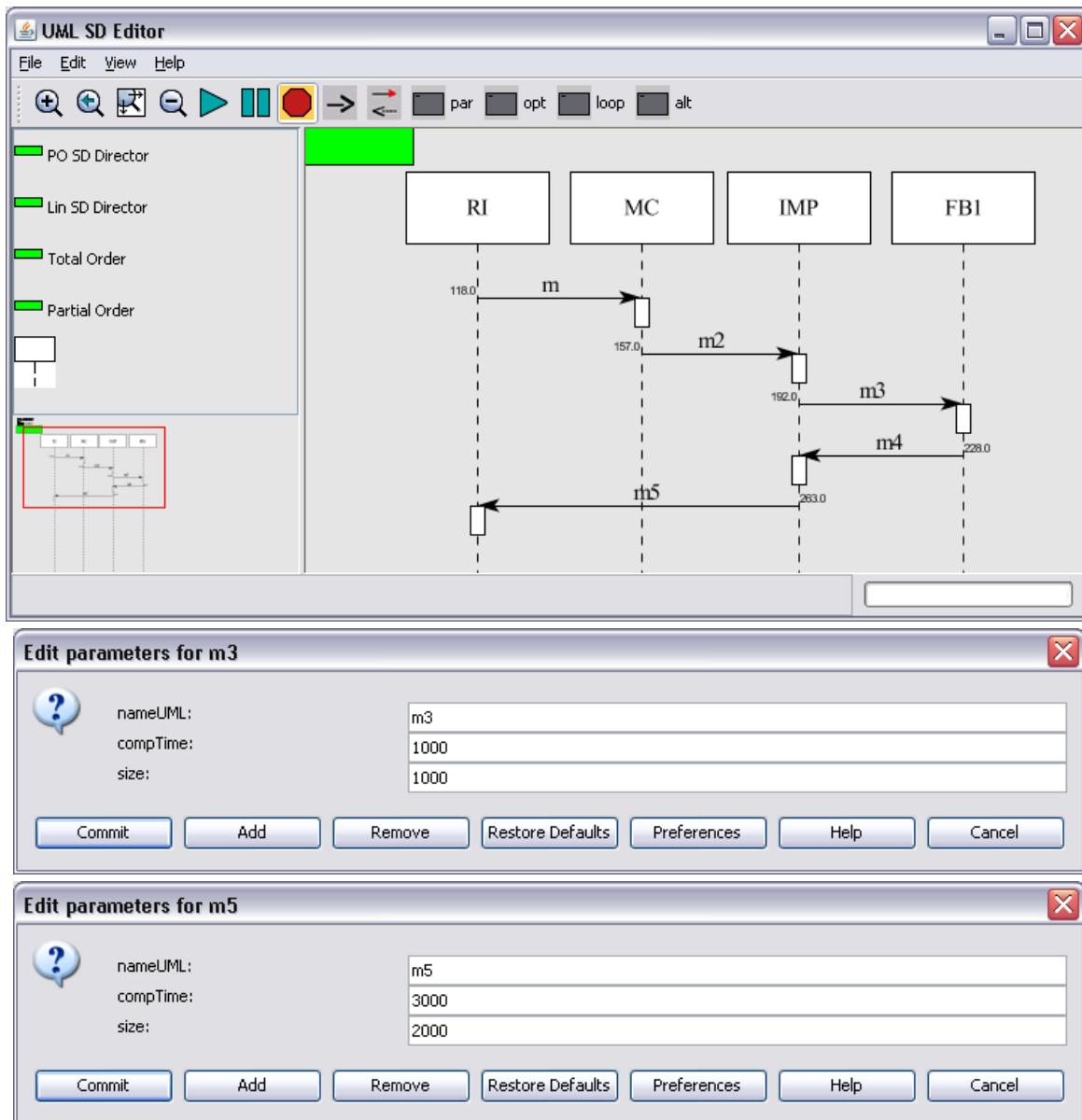


Fig. 3.8: Computation time is not binded to a task, but to messages sent by one task. The upper window shows a sequence diagram and the messages exchanged by its corresponding tasks. The two lower windows show the computation time (compTime) and the message size of two messages sent by task 'IMP'.

An application task can be mapped to any type of IP core, provided its size, usage and efficiency for that particular type of IP core is specified to be greater than zero. Using

these parameters, the mapper model can make more realistic mapping decisions. Fig. 3.9 shows parameters of an application task 'IMP' (Image Processor) for a heterogeneous system, showing its type compatibility to the IP core of type '0' with its size of '65', utilization of "100%" and affinity of "20%" and type compatibility to the IP core of type '1' with its size of "100", utilization of "20%" and affinity of "100%". Please note that the parameters 'utilization' and 'affinity' are divided by 100 before using on Eq. 3.1 and Eq. 3.2.

Parameter	Value
sizeOfType0:	65
sizeOfType1:	100
sizeOfType2:	0
sizeOfType3:	0
sizeOfType4:	0
utilizationOfType0:	100
utilizationOfType1:	20
utilizationOfType2:	0
utilizationOfType3:	0
utilizationOfType4:	0
affinityOfType0:	20
affinityOfType1:	100
affinityOfType2:	0
affinityOfType3:	0
affinityOfType4:	0

Fig. 3.9: Parameters of an application task 'IMP'.

3.2.3 Multi-Task Manager

Having the required 'utilization' of each task calculated by Eq. 3.2, it is now possible to give support to multi-tasking on each IP core of the MPSoC. The utilization of an IP core mapped with multiple tasks can be calculated by

$$U_k = \sum_{t=0}^q U_t \quad \text{Eq. 3.3}$$

where U_k is the utilization of a processor k , q is the amount of tasks mapped to k and U_t is the utilization of each task t mapped on k presented on Eq. 3.2. With Eq. 3.3 it is now possible to know if an IP core is overloaded when its utilization surpasses the 100%. When this happens, the IP core will start delaying equally all tasks in an attempt to provide the best possible service for each task. Thus, the current implementation of the multi-task manager provides no real time services, due to its simple round-robin scheduling algorithm. For this reason it is possible to provide a maximum utilization for each IP core and a maximum number of tasks that can be mapped to one IP core, as illustrated on Fig. 3.10.

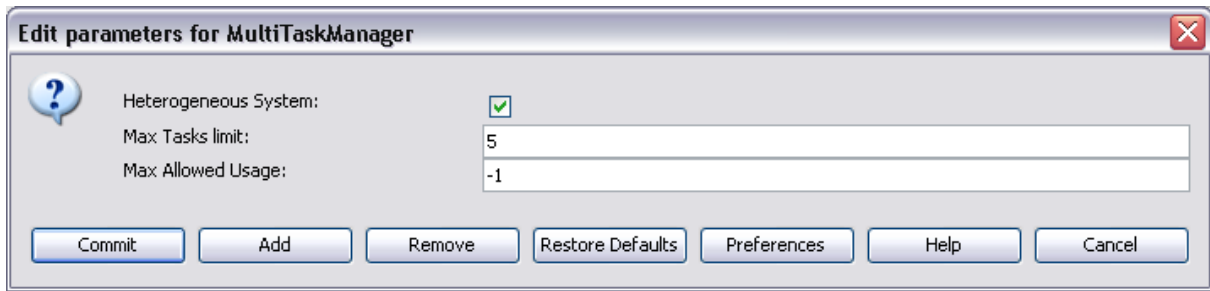


Fig. 3.10: Parameters of the Multi-Task Manager.

3.2.4 UsageScope

As mentioned on Section 2.3, the platform model uses the NoCScope to facilitate system's evaluation and optimization. After the introduction of the type system and consequently the utilization parameter for application actors, a new scope called UsageScope was implemented and added to the NoCScope framework. The UsageScope is illustrated on Fig. 3.11 and it can be useful for having a visual feedback of the utilization of every IP core of the platform. This scope can also be used by task mapping and task migration algorithms for not overloading an IP core and for load balancing techniques.

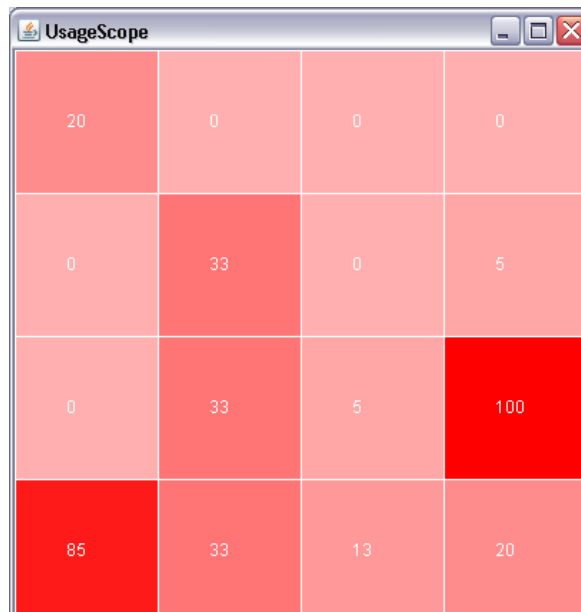


Fig. 3.11: Snapshot of the UsageScope. Each square presents the percentage instantaneous utilization of one IP core of the platform.

3.3 Summary

This Chapter presented different application modeling alternatives that support the joint validation of application and the platform models presented on Chapter 2. The approach is based on executable models, and it uses a back-annotation strategy to increase the accuracy of the execution of a given application by considering the timing behavior obtained from the platform on which the application runs.

Applications can be modeled according to three different schemes: synthetic traffic, instruction set simulator (ISS) and UML sequence diagrams. While the synthetic traffic scheme can easily express the communication behavior of applications that were already simulated/executed and had their traces captured, it lacks the computation behavior of the application. The ISS scheme provides a full coverage for modeling applications with regard to computation and communication, but it is far too complex to instantiate several ISSs on a MPSoC system for simulation purposes. The UML sequence diagrams scheme is a more appropriate solution that can consider the computation and communication aspects of applications with a more abstract description. Therefore, the UML sequence diagrams scheme is the only scheme used from this point on of this thesis with the goal to evaluate task mapping and migration techniques.

The MARS ISS that was connected to the RENATO NoC model could be integrated to the UML sequence diagram scheme if the goal of the MPSoC developer is to achieve an intermediate solution between an abstract modeling and accuracy, with the clear expense on simulation time. This could be accomplished since the network interface that interconnects the ISS to the platform encapsulates all processor communications into tokens, which is the only requirement for implementing application actors as described on Section 3.1.3.

4 Task Mapping and Migration[‡]

Applications running on Multiprocessor Systems-on-Chip (MPSoCs) may vary dynamically at execution time according to user (e.g. load of new applications) and/or performance (e.g. change the frequency operation for optimizing battery lifetime) requirements, which leads in both time-changing processor workload and communication patterns [43][44][45]. Thus, offline-mapping techniques can be sub-optimal or inadequate in many scenarios. In this context, dynamic task mapping techniques have been used to achieve the required runtime adaptability demanded by such multiprocessing systems [46][47]. Such dynamic task mapping techniques are evaluated in both homogeneous and heterogeneous platform architectures.

More than avoiding congestion and placing communicating tasks near to each other, heterogeneous MPSoCs need to care about the affinity of tasks with the IP cores available on the platform. This is only true when the same task is developed for different IP cores and trading efficiency against utilization of these cores is left for the system to balance. The result is then a computing system that can analyze its own resources and allow their use in a more optimized manner. Therefore, smart implementations of dynamic mapping algorithms are vital for the MPSoC to execute applications with good performance figures and using as few resources as possible.

The goal of this Chapter is to present different task mapping and migration algorithms and to evaluate them quantitatively and comparatively on a model that considers them jointly with the application and the heterogeneous platform presented on Chapters 2 and 3. Some of the presented algorithms are multi-objective, considering not only the affinity of the task and the congestion of the network, but also the utilization of the IP cores, the position on the network and the amount of communication among tasks. In order to have an implementation of such algorithms, two additional elements were introduced to the approach presented on the previous Chapters: the mapper actor and the task migration actor. Both have access to information from both the application and platform models. While the mapper reacts to events that the directors of sequencing actors and the abstract processing elements generate, the task migration actor is a standalone entity that wakes up at regular intervals of time to verify if all tasks are mapped to an optimum position.

[‡] Major parts of this Chapter were submitted to DATE [115].

4.1 Related Works

Examples of dynamic task mapping techniques explored in homogeneous architectures are [45][46][48]. In turn, dynamic task mapping on heterogeneous MPSoC platforms are investigated in [43][49][44][50][51][52][53]. Due to the distinguish nature of IP cores that can incorporate such platforms, the mapping process is more complex when compared to homogeneous scenarios, since heterogeneous constraints must be considered at run-time. In this context, Carvalho et al. [43] proposed and evaluated the performance of six mono-task mapping heuristics considering different application workloads. Some of these heuristics were extended to consider multi-tasks mapping onto the same IP core, while minimizing the commutation overhead in the same NoC-based platform [49]. Singh et al. [49] also proposed new heuristics that consider the power consumption as the product of number of bits to be transferred and distance between source-destination pair.

Kempf et al. [54] present a framework targeted to MPSoC software development, verification, and evaluation. Their framework does not require the platform model to be complete before the software development can start. Software can be developed in four different levels of abstraction that vary in accuracy and simulation speed. The framework uses SystemC for simulation and XML to describe task mappings and timings. Furthermore, the framework provides an efficient design space exploration environment for instance by providing designers with various communication architectures.

Faruque et al. [44] present a distributed agent-based mapping scheme. The proposed scheme divides the system into virtual clusters. A cluster agent (CA) is responsible for all mapping operations within a cluster. Global agents (GAs) store information about all the clusters of the NoC and use a negotiating policy with CAs in order to define to which cluster an application will be mapped. Another distributed approach is proposed in [45], which explores different implementations of a decentralized self-embedding algorithm, aiming to minimize network contention and latency while providing fault-tolerance support for NoC-based systems.

Ristau et al. [55] discuss design space exploration early at the design process as well as the exploration of different mapping strategies. However, their application and platform models are simplified, disregarding the inter-process communication costs. Lei & Kumar [56] describe the application as parameterizable task graphs, which are mapped onto NoC architecture. Their work aims at supporting mapping based on genetic algorithms and minimizing overall execution time of a task graph. Furthermore, they also implemented a tool that uses a two-step genetic algorithm, achieving optimal solutions for regular NoC topologies in affordable time.

In [50], a run-time spatial mapping technique with real-time requirements that considers streaming applications is investigated. The application mapping is determined according to a set of information (i.e. latency/throughput) that is collected at design time, aiming to satisfy the QoS requirements, as well as to optimize the resources usage

and to minimize the energy consumption. Smit et al. [51] present an iterative hierarchical mapping strategy that tries to minimize the energy consumption of the MPSoC while providing QoS. Ferrandi et al. [52] introduce an ant colony optimization heuristic, which executes both scheduling and mapping in order to optimize the application performance. A similar approach that uses scheduling information/constraints in order to achieve a better mapping decision at run-time is proposed in [53].

Riccobene et al. [57] present a System-on-Chip (SoC) design methodology using code generation from UML diagrams to an executable SystemC model. Furthermore, Arpinen et al. [58] use UML state chart diagrams to support automatic code generation in order to map UML applications onto the platform.

4.2 Initial Task Mapping

In contrast to static task mapping, dynamic task mapping does not need to know all tasks of the applications before the execution of the system. However, the dynamic mapping algorithm can perform better mapping decisions when it knows at least which tasks of the application communicate more to which others. Without this information the mapper has no means of knowing if a task should be mapped to a more centralized position of the network (where more tasks can be later on mapped around this task and thus perform more parallel communications at a short distance) or to a corner position (where less neighbors exist). This problem is intensified in the beginning of the execution of the application since most of the dynamic mapping algorithms use the distance factor on their cost functions, meaning that a bad mapping of the initial tasks will affect all later dynamic mapping decisions.

Another situation, which is aggravated in the beginning of the execution of the application, is that most of the platform resources are free and waiting for tasks to start computing and communicating. If most of the resources are free, the dynamic mapping algorithm cannot rely also on the used resources of the platform as a metric for deciding where to map a task.

Due to these limitations related to the initial tasks of the application, the mapping process was separated in two phases: Initial Mapping and Dynamic Mapping. In the first phase only the initial task of each sequence diagram of the application is mapped. The other tasks are mapped after required in the dynamic mapping phase, as illustrated by the data flow diagram presented on Fig. 4.1. Fig. 4.2 shows the class diagram of the First Free (FF) and Cluster (CL) initial mapping algorithms used on this work, which are both subclasses of the class "InitialMapper".

4.2.1 First Free (FF)

The First Free (FF) initial mapping algorithm simply selects the next compatible IP core to map a given task, thus walking sequentially through all IP cores before considering an IP core again. The IP cores are checked row by row from left to right, and the first one

which is compatible and can receive the initial task is chosen for mapping. Fig. 4.3 presents four initial tasks (T1-T4) used by four different sequence diagrams mapped on the platform. The arrow shows the order on which the IP cores are visited by the FF algorithm.

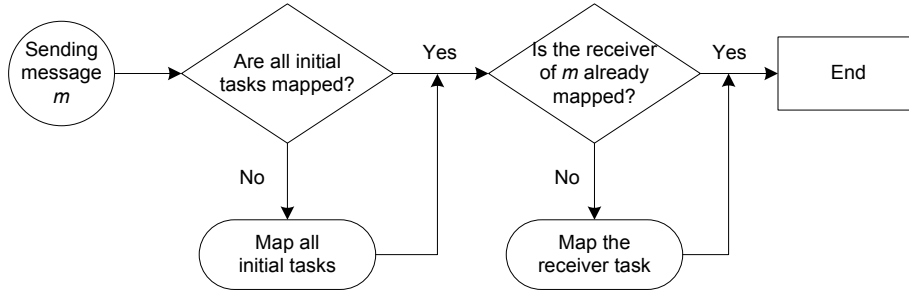


Fig. 4.1: Data flow diagram of task mapping.

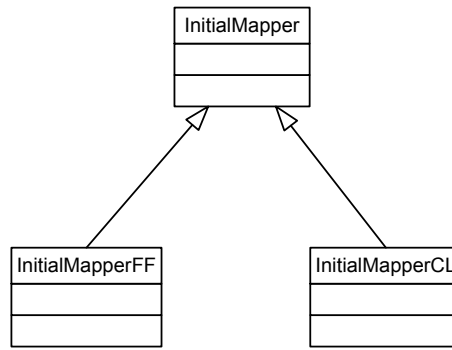


Fig. 4.2: Class diagram of the initial mapping classes.

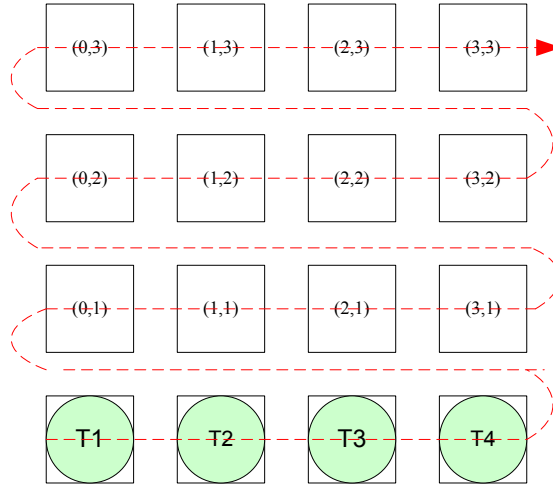


Fig. 4.3: First Free (FF).

4.2.2 Cluster (CL)

The Cluster (CL) initial mapping algorithm separates the IP cores in clusters and maps the initial tasks of different sequence diagrams on different clusters. The size of the clusters depends on the amount of initial tasks and on the size of the network. Fig. 4.4 shows 9 possible 2x2 clusters on a 4x4 mesh network. When there is up to four initial tasks, the clusters presented on the left are used. If more than four initial tasks exist, the

five other clusters presented on the right side of the figure are also employed.

The cluster which contains the IP core (0,0) is always the first cluster to be chosen for mapping an initial task. After that, the cluster with less tasks not belonging to the same sequence diagram is chosen for mapping. This enforces initial tasks of different sequence diagrams to be separated on the network and the IP cores around them to be free for other tasks belonging to the same sequence diagram that will be mapped later. The result is a reduced number of channels reserved for data transmissions and reduced energy consumption.

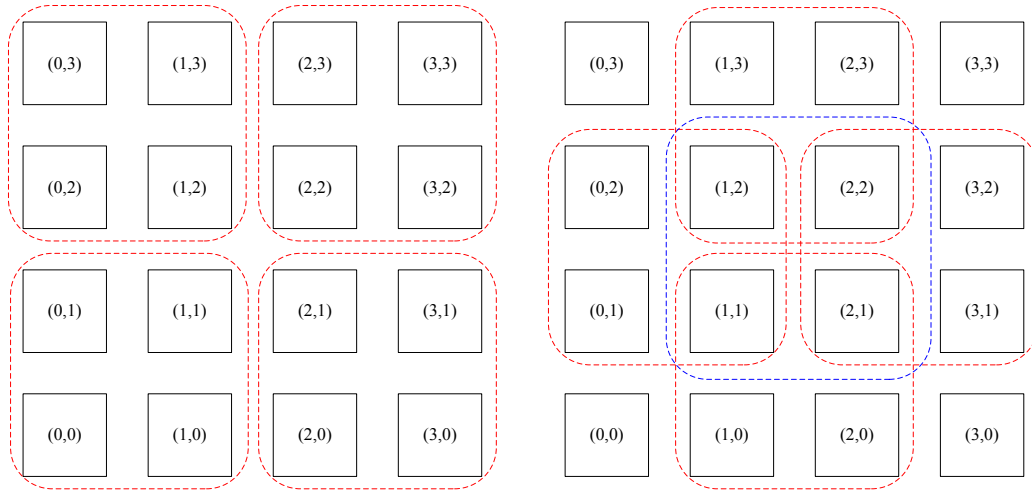


Fig. 4.4: Different 2x2 clusters on a 4x4 mesh network.

4.3 Dynamic Task Mapping

After the initial tasks are mapped, the following tasks are mapped on demand as soon as a message is about to be sent into the network. This guarantees that a task will not use resources of the platform if it is not really going to execute.

On an effort of creating an extensible library of dynamic mapping algorithms, the following Sections describe eight different dynamic mapping algorithms. The class diagram illustrated on Fig. 4.5 shows that all dynamic mapping algorithms extend the super class “DynamicMapper”, which provides methods that are common to all subclasses.

Independently of which dynamic mapping algorithm is chosen, the mapper is responsible to assign tasks/lifelines to abstract IP cores at the platform model. For each token a sequencing actor receives, a given message within its sequence diagram will be triggered (for example, the message M1 sent by lifeline L1 to lifeline L2 on Fig. 3.5). When this happens, the corresponding director D2 interrupts the delivery and notifies the mapper about the message. Since the mapper is responsible for assigning each lifeline to an abstract IP core, it knows that for instance lifeline L1 is mapped to IP core 2, whereas L2 is mapped to IP core 7. Once the mapper receives the information about the triggered message, it will command the IP core associated to the sender of the message (IP core 2) to generate the corresponding traffic into the interconnect structure (in the

case of a NoC platform, it must create a packet with destination, size, and payload and write this packet on the local input port of the corresponding router). Then the mapper waits until the processing element associated to the receiver of the message (IP core 7) notifies the complete reception of the packet. Upon notification, the mapper calls back the director D2 (which has notified the triggering of the message) and informs it that the message can now be delivered. After that, the director can forward the message to the output port of the sequencing actor, and the message reaches its destination with the exact latency that it would take if the application is executed on top of the implementation platform [112].

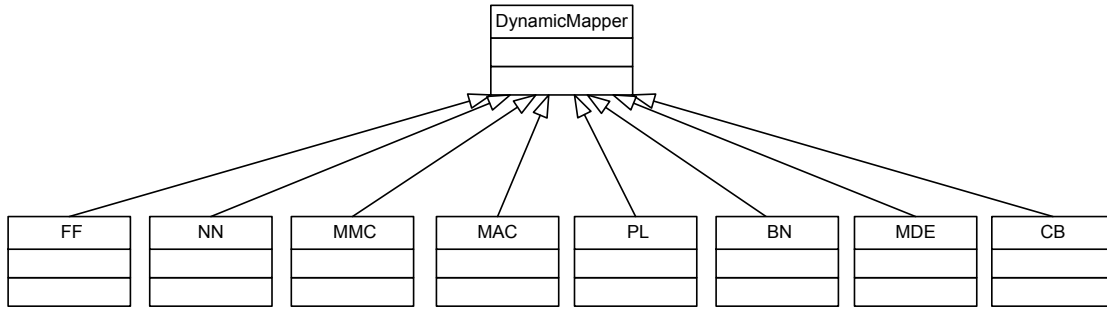


Fig. 4.5: Class diagram of the dynamic mapping algorithms.

4.3.1 First Free (FF)

The initial mapping algorithm FF already described on Section 4.2.1 can also be used at runtime for the non-initial tasks. Fig. 4.6 shows the data flow diagram of the FF dynamic mapping algorithm.

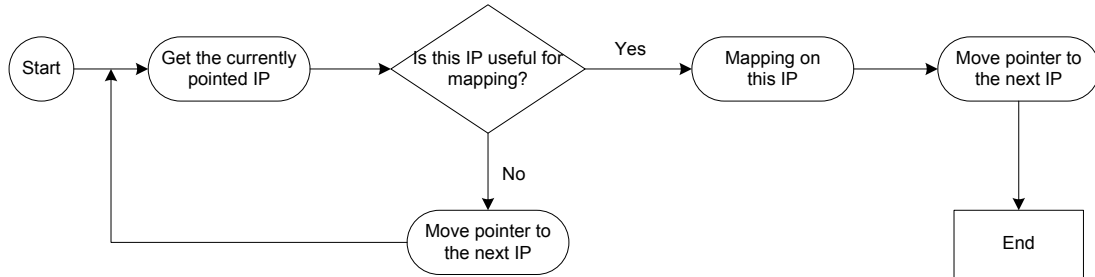


Fig. 4.6: Data flow diagram of the FF dynamic mapping algorithm.

4.3.2 Nearest Neighbor (NN)

The Nearest Neighbor (NN) dynamic mapping algorithm finds an IP core for the required task which is nearest to the IP core where the sender is mapped. It begins checking all the IP cores at 1-hop distance of the sender. The first IP core that can receive the task is chosen for mapping. If an IP core candidate is not found at 1-hop distance from the sender, the IP cores at 2-hop distance are checked and so on, as illustrated by Fig. 4.7 with the sender task exemplified in the middle of the network. Using this method, tasks that communicate with each other may be mapped very closely and therefore the communications between them take less time and energy. Fig. 4.8 shows the data flow of the implementation of Nearest Neighbor.

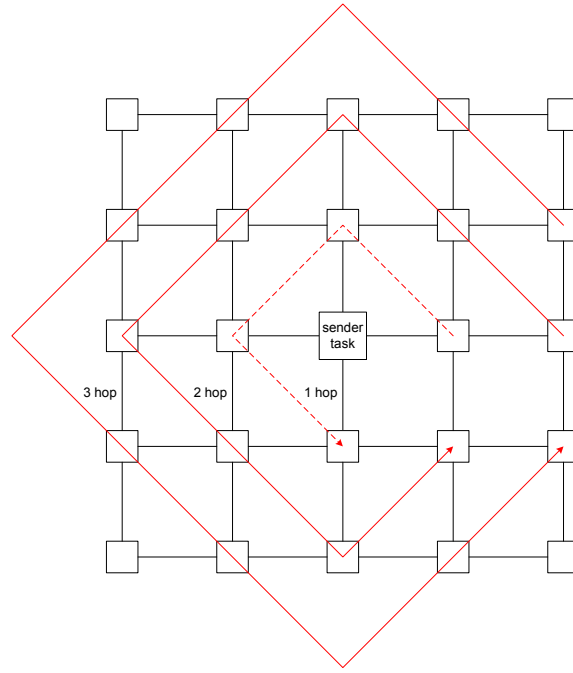


Fig. 4.7: Order of visiting IP cores according to the NN dynamic mapping algorithm and considering that the sender task is mapped on the center of the network.

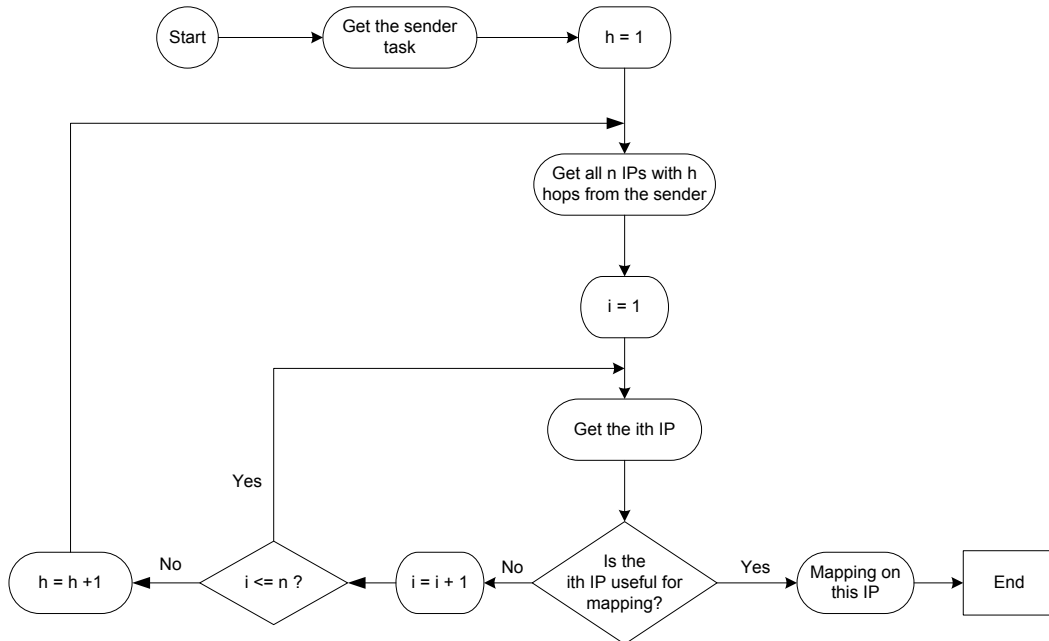


Fig. 4.8: Data flow diagram of the NN dynamic mapping algorithm.

4.3.3 Minimum Maximum Channel Load (MMC)

Minimum Maximum Channel Load (MMC) is a congestion aware mapping algorithm created by Carvalho et al. [59]. The main goal of this algorithm is to reduce the maximum occupation of the network channels and so to avoid congestion on the network. For Carvalho et al. every message between two tasks is defined by volume (amount of data sent from one task to another) and message rate (the rate is expressed as a percentage of the available channel bandwidth). On this work the volume can be

directly extracted from the message size used on the sequencing actors, while the message rate can be calculated by

$$R_m = \frac{V_m \times 100}{P_m \times TBC} \quad \text{Eq. 4.1}$$

where, R_m is the rate of message m , V_m is the volume of message m , P_m is the period of message m (how frequently message m is sent) and TBC is the total bandwidth of a channel (expressed with the same metrics used by the ratio V_m / P_m). P_m is the same for every m of the same sequencing actor and can be extracted from the initiator actor of the sequencing actor where m is defined. TBC can be gathered from the flit size of the chosen NoC used as a platform. Then the utilization of a channel can be calculated by

$$U_c = \sum_{m=0}^n R_m \quad \text{Eq. 4.2}$$

where U_c is the utilization of a given channel, n is the amount of messages passing through this channel c , R_m is transmission rate of message m . The data structure used to hold the utilization of every channel of the network is a hashtable, which is named “IP_neighborIP_utilization” and is illustrated on Fig. 4.9. The IP cores are the key of this hashtable and the value is another hashtable called “neighborIP_utilization”, which uses the neighbor IP cores as keys and the utilization of the channel as values. This hashtable needs to be updated every time a task is mapped in or out of the system. Another situation where the hashtable requires modification is if a task migrates inside the system, as presented on Section 4.5.

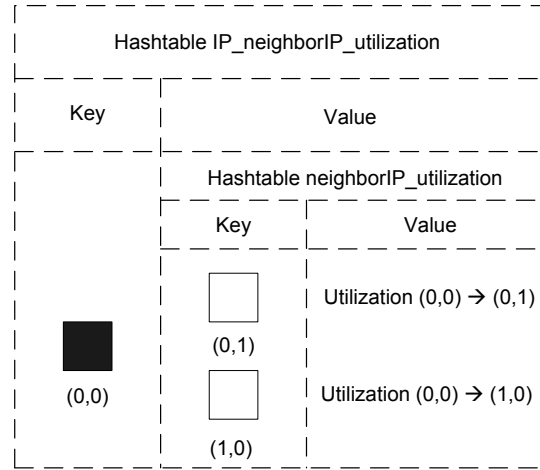


Fig. 4.9: Data structure that holds the utilization of every channel of the network.

With the information provided by the “IP_neighborIP_utilization” hashtable, the MMC algorithm computes the maximum channel utilization of each possible mapping and the one with the minimum value is chosen. Fig. 4.10 presents an example of the channels utilization of a 4x4 network. The current maximum utilization of the network is 40. And now the task MC needs to send a message with a utilization of 30 to the task DC. Task DC is not mapped yet and it can be mapped on 7 IP cores ((1,1), (3,1), (2,2), (0,3), (1,3), (2,3) and (3,3)). Fig. 4.11 shows the maximum channel utilization of 2 possible task

mappings, which are on IP cores (0,3) and (1,1). The mapping (a) has a maximum channel utilization of 40 and the mapping (b) has a maximum channel utilization of 70. The other 5 possible mappings have the same maximum channel load like mapping (b). So, mapping (a) is chosen for mapping task DC.

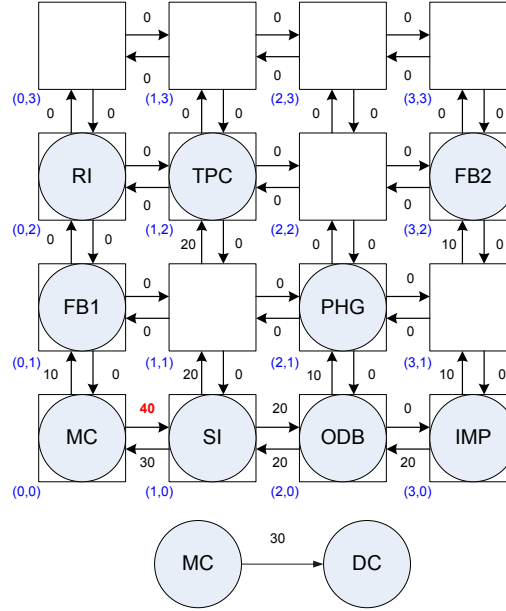


Fig. 4.10: Network utilization before mapping.

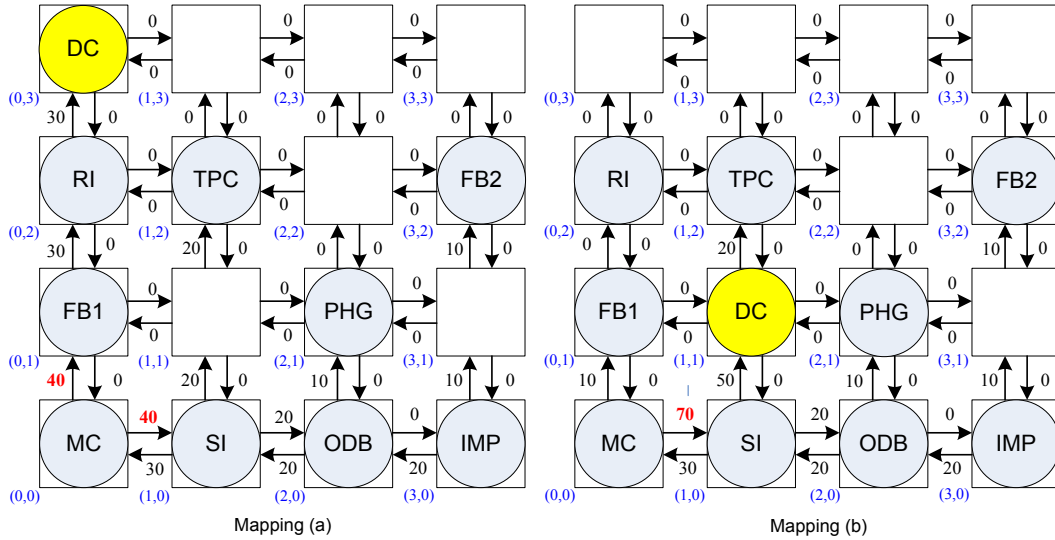


Fig. 4.11: Two places where the task DC can be mapped and the corresponding new utilization of the channels according to the MMC dynamic mapping algorithm.

Fig. 4.12 shows the data flow diagram of the MMC algorithm. Initially the current maximum channel utilization of the network is calculated. Then a loop is built, in which each possible mapping is evaluated with the MMC algorithm. The maximum channel utilization of each possible mapping is calculated and compared to the minimum maximum utilization found. The mapping which has the minimum maximum channel utilization is chosen for mapping the required task.

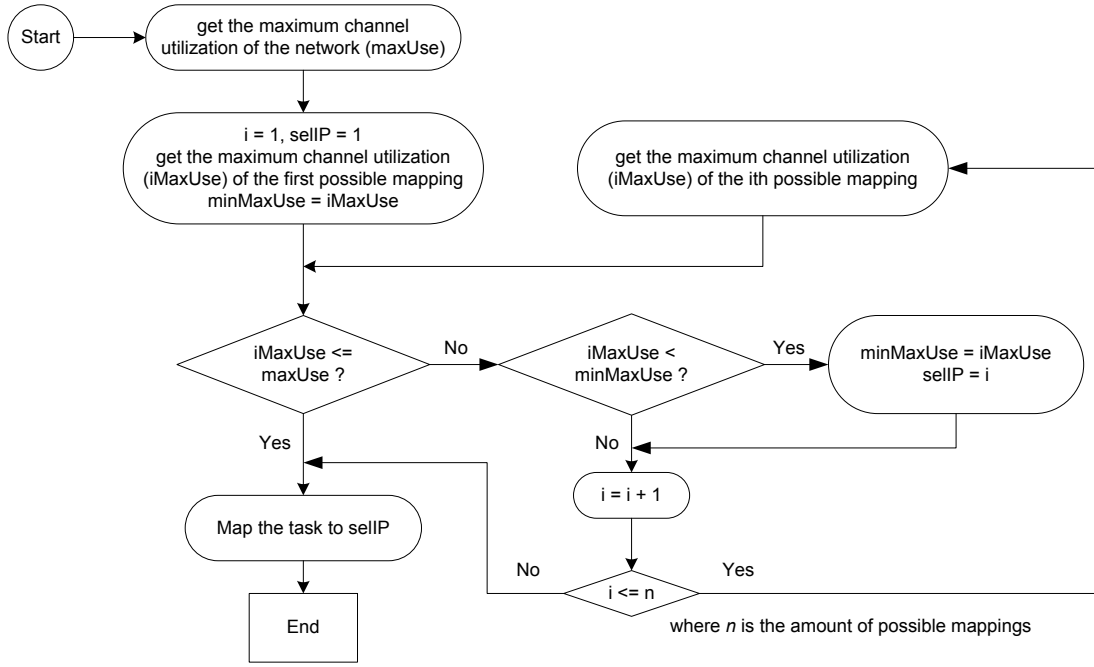


Fig. 4.12: Data flow diagram of the MMC dynamic mapping algorithm.

4.3.4 Minimum Average Channel Load (MAC)

The Minimum Average Channel Load (MAC) is another congestion aware algorithm introduced by Carvalho et al. [59]. It calculates the average channel utilization of the whole network and selects the mapping solution that has the minimum average channel utilization. The MAC algorithm is very similar to MMC and is presented on Fig. 4.13. The average channel utilization is calculated by

$$Avg = \frac{\sum_{i=0}^c U_i}{c} \quad \text{Eq. 4.3}$$

where c is the number of channels that exists on the network and U_i is the utilization of the channel i calculated by Eq. 4.2. Using network presented on Fig. 4.10 as a starting point and applying the average equation presented on Eq. 2.2, the average utilization before mapping task DC is 4.375 (210/48). Fig. 4.14 presents two possible mappings for task DC. On mapping (a) the message from MC to DC passes through the routers (0,0), (0,1), (0,2) and (0,3), and the new total average utilization is 6.2 (300/48). On mapping (b) the message from MC to DC passes through the routers (0,0), (1,0) and (1,1), and the new total average utilization is 5.6 (270/48). The average of each other possible mapping is calculated in the same way. They are all bigger than mapping (b) and therefore mapping (b) is chosen in this case.

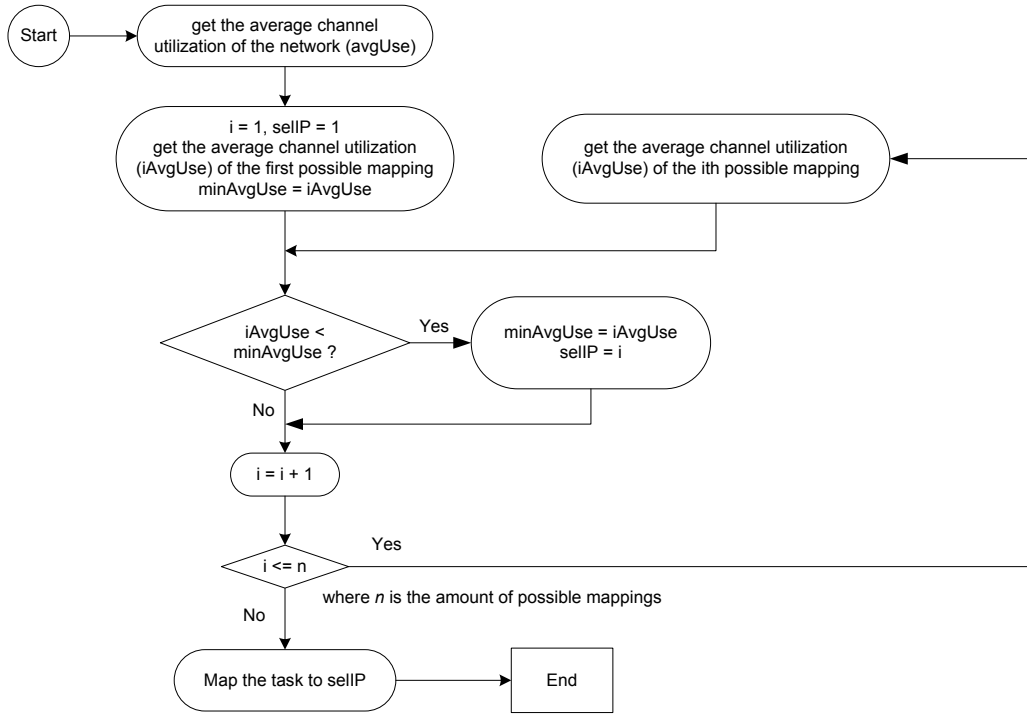


Fig. 4.13: Data flow diagram of MAC.

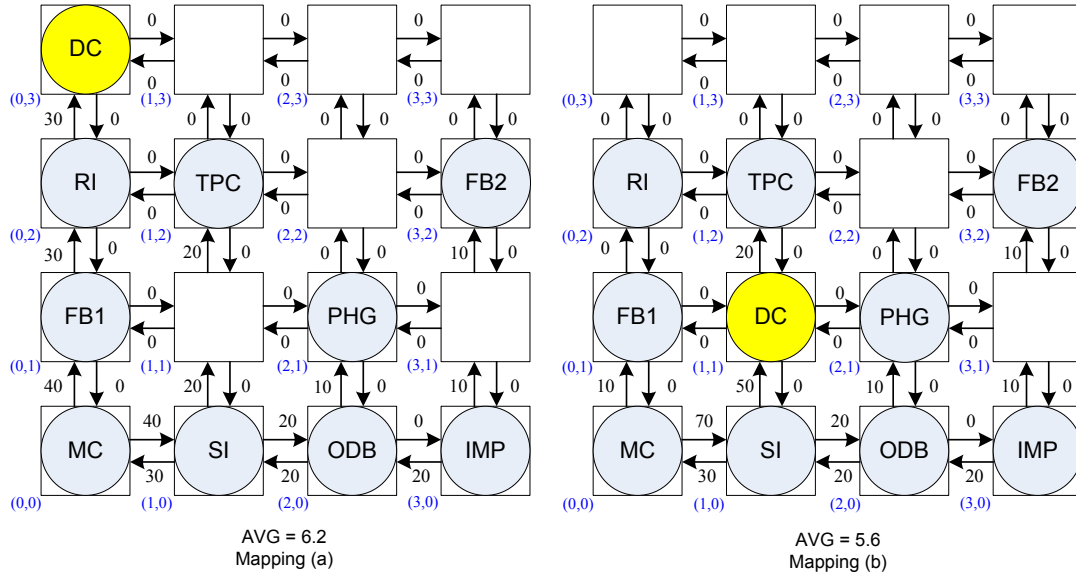


Fig. 4.14: Two places where the task DC can be mapped and the corresponding new utilization of the channels according to the MAC dynamic mapping algorithm.

4.3.5 Path Load (PL)

The Path Load (PL) algorithm is another algorithm created by Carvalho et al. [59] that considers the channel utilization not on the whole network, but in individual channels between the sender task and the new receiver, thus reducing the execution time of the mapping algorithm if compared to MMC or MAC. PL employs the sum of the utilization of the channels between sender and receiver as cost function of the algorithm, and the

mapping that presents the minimum sum is selected. The PL algorithm is illustrated in the data flow diagram in Fig. 4.15. Considering the network presented on Fig. 4.10 as a starting point, Fig. 4.16 presents two possible mappings of task DC following the PL algorithm. On mapping (a) the message from MC to DC passes through routers (0,0), (0,1), (0,2) and (0,3), and the path utilization is 100 (40+30+30). On mapping (b) the message passes through the routers (0,0), (1,0) and (1,1), and the path utilization is 120 (70+50). The other possible mapping solutions have all bigger path utilization than mapping (a). Therefore mapping (a) is chosen in this case.

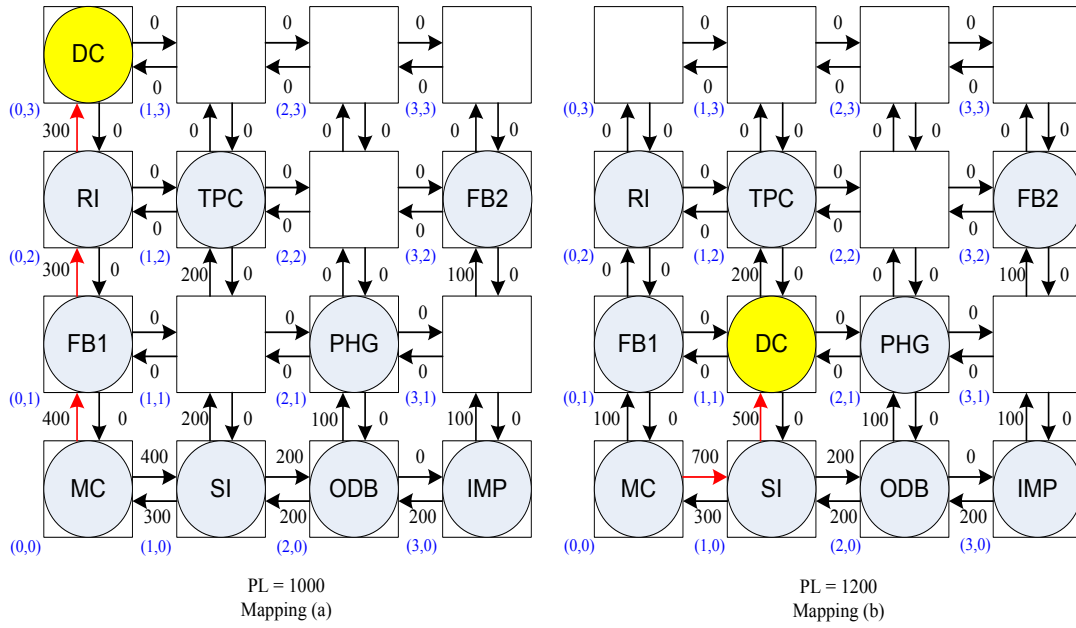


Fig. 4.15: Data flow diagram of the PL dynamic mapping algorithm.

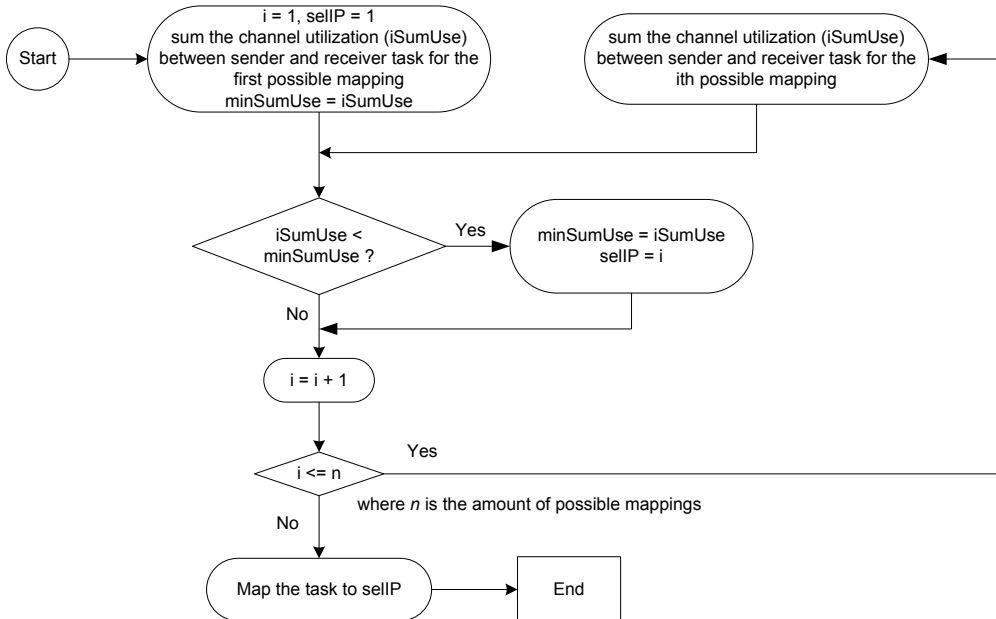


Fig. 4.16: Two places where the task DC can be mapped and the corresponding new utilization of the channels according to the PL dynamic mapping algorithm.

4.3.6 Best Neighbor (BN)

Best Neighbor (BN) algorithm is a combination of NN and PL, which is also an algorithm proposed by Carvalho et al. [59]. Like PL it selects the mapping solution that has the minimum path utilization. But differently from PL it computes not all possible mapping solutions, but only the mapping solutions that are closer to the sender task (like NN). So, if there is any available IP cores with h -hop distance from the sender, the IP cores in distance of $h+1$ -hop are not tested. On the one hand that this reduces the computation time for executing the mapping algorithm, on the other hand some good possible solutions for mapping at one hop away can be neglected.

As shown in the data flow diagram in Fig. 4.17, the implementation of BN uses two loops. The first loop implements the search way of NN algorithm and find all available IP cores with same hops from the sender. The second loop implements the selection method of PL algorithm and decides the final mapping solution.

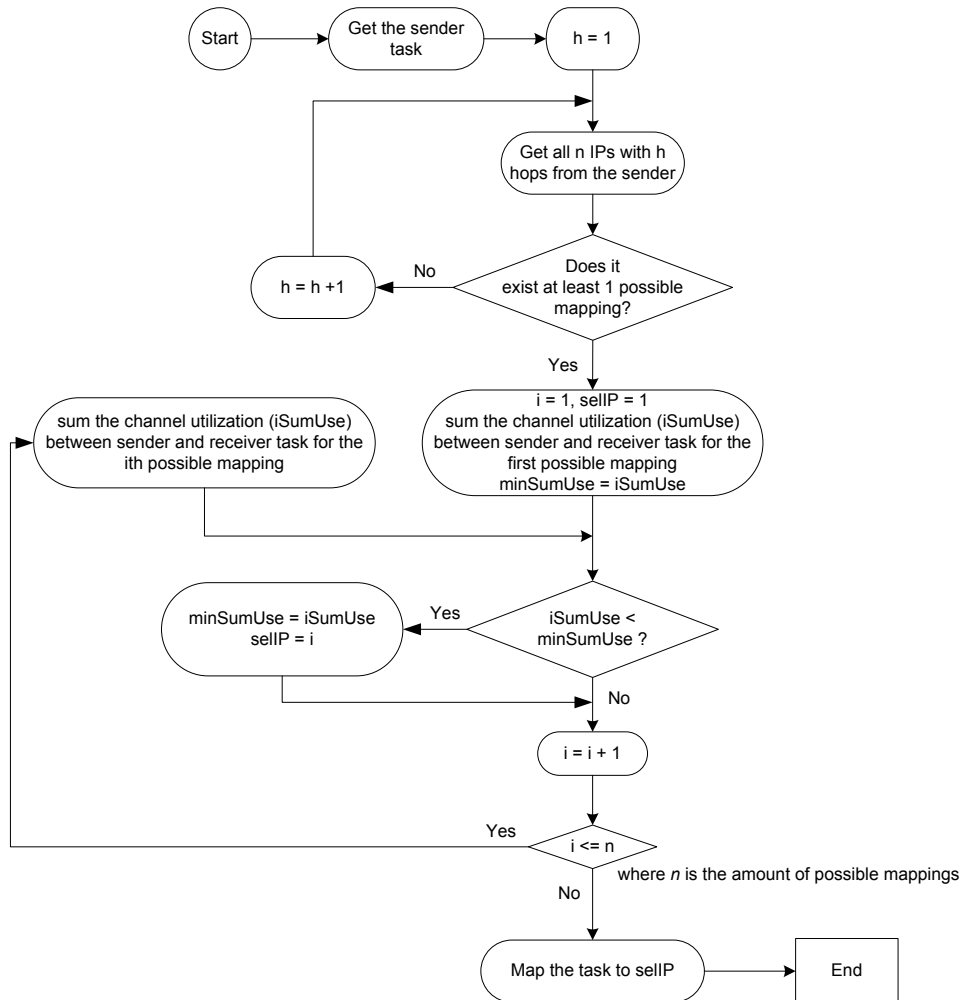


Fig. 4.17: Data flow diagram of the BN dynamic mapping algorithm.

4.3.7 Minimum Data Exchange (MDE)

The Minimum Data Exchange (MDE) considers all possible mappings for a given task

and computes for each of them the total amount of data that must be sent/received by the already mapped tasks. The IP core with less communication volume receives the target task. If more than one IP core returns the same communication volume (very likely to happen in the beginning of the execution of the system), the IP core with minimum hops distance to the requesting task is selected. If again there is more than one candidate IP core, the first candidate of an array of final candidates is selected. Fig. 4.18 illustrates the algorithm of the MDE with a data flow diagram.

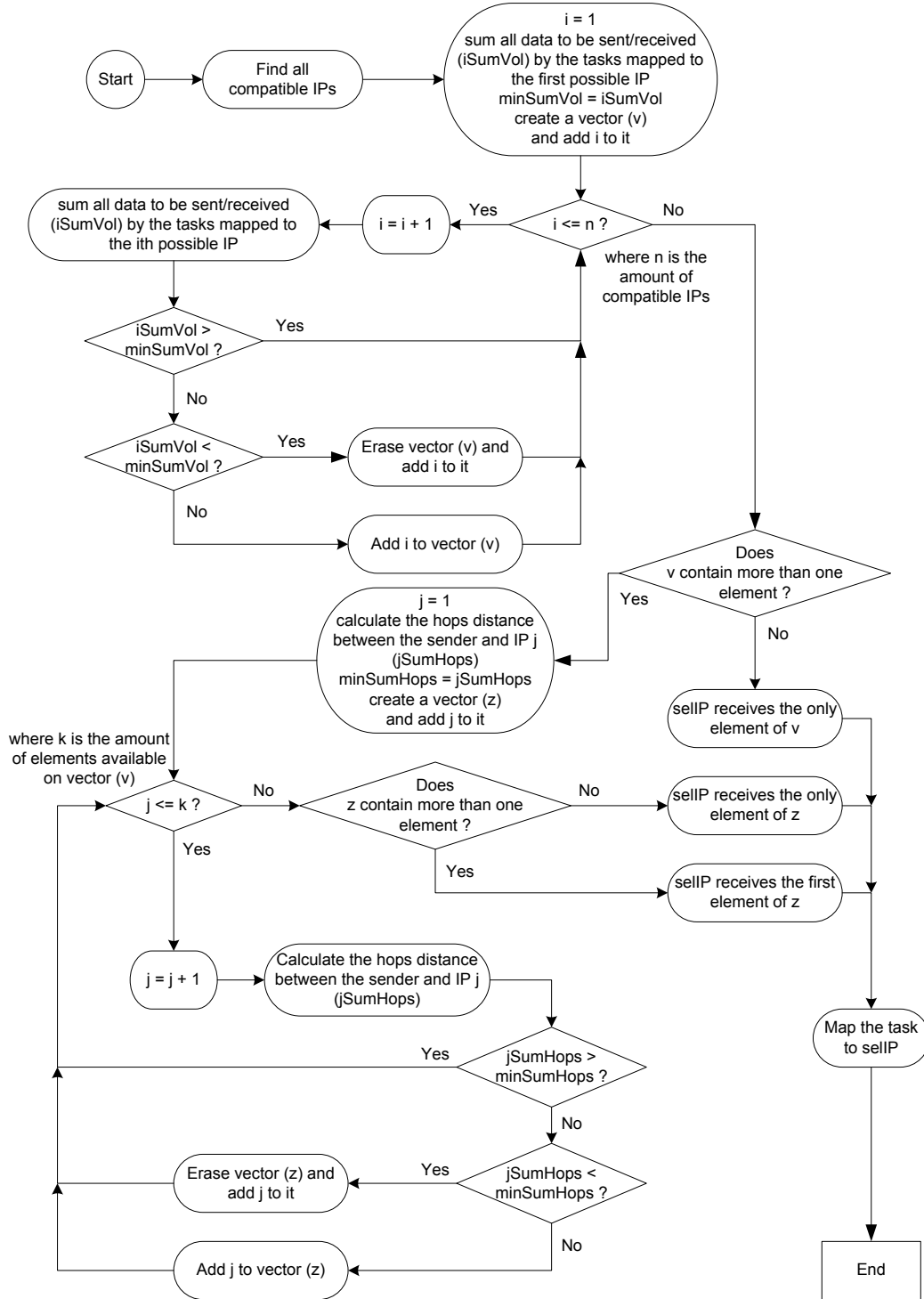


Fig. 4.18: Data flow diagram of the MDE dynamic mapping algorithm.

4.3.8 Cost Based (CB)

None of the previous dynamic mapping algorithms consider the computation time of the tasks or the affinity of the task to a certain IP core. The main goal of this algorithm is to take these parameters into consideration together with the communication volume of the tasks and the number of hops of the communications. With this goal in mind the CB considers all possible mappings for a given task and chooses the one with minimum cost according to the following equation

$$Cost = \frac{U_k \times H_{sr} \times V_{sr}}{Af_{rk}} \quad \text{Eq. 4.4}$$

where U_k is the current utilization of the IP core under consideration for mapping k (presented on Eq. 3.3), H_{sr} is the number of hops between the sender task s and the receiver task r (considering r mapped on k), V_{sr} is the volume between s and r measured by the amount of bytes exchanged by them, and Af_{rk} is the affinity of the target task r to k . Each of these parameters used on Eq. 4.4 can be included or excluded of the cost function of the CB algorithm using parameter selectors provided in the workspace, as presented on Fig. 4.20. Taking the hop-volume into account ensures that heavily communicating tasks are mapped close together. Including affinity on the cost function ensures that application tasks are mapped on the most efficient IP cores, consuming less computation time. Considering the utilization avoids overloading an IP core of the system and provides a balanced system in terms of IP cores usage. Fig. 4.19 illustrates the algorithm of the CB with a data flow diagram.

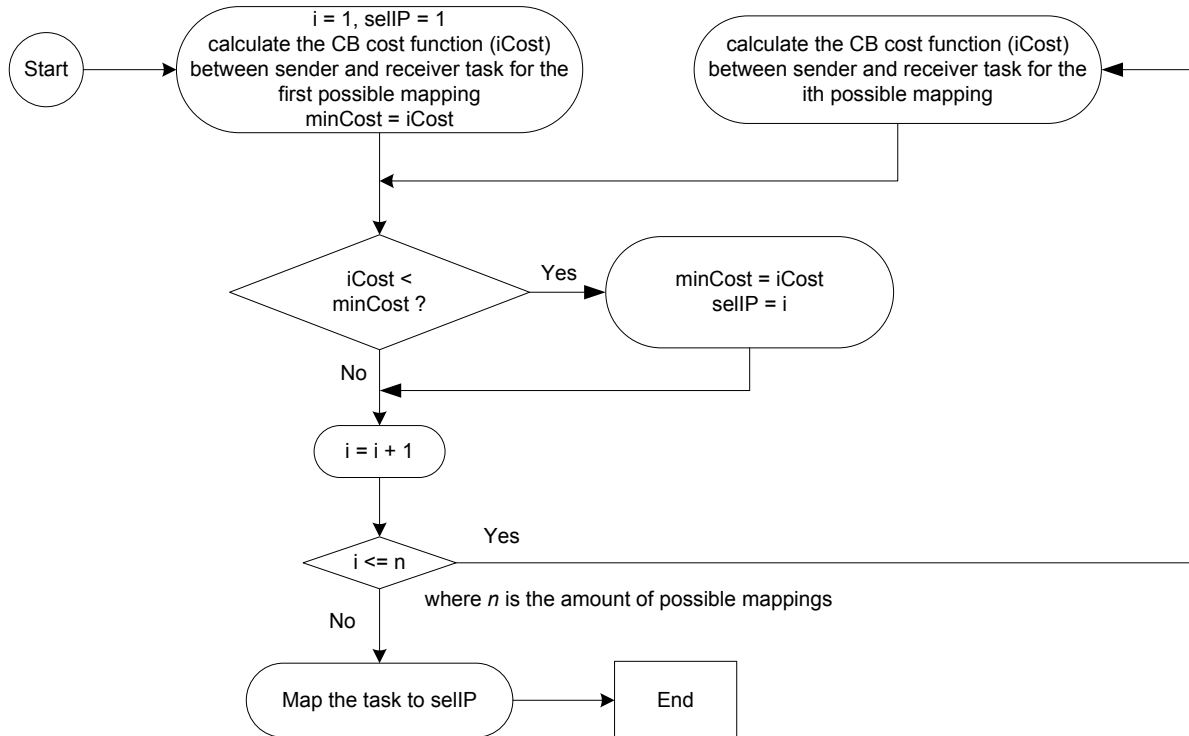


Fig. 4.19: Data flow diagram of the CB dynamic mapping algorithm.

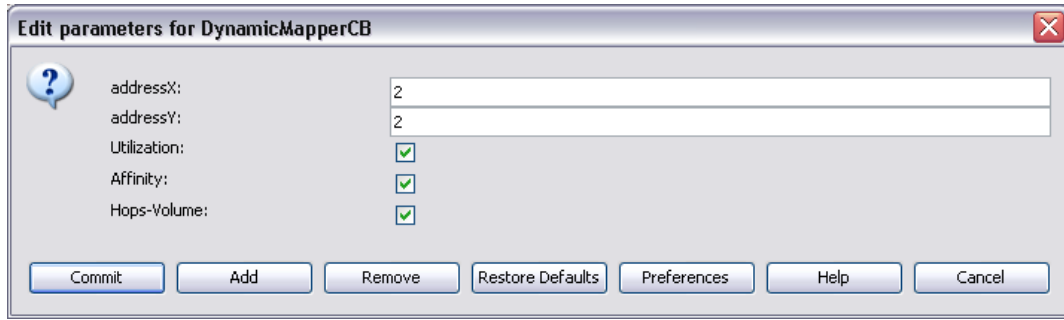


Fig. 4.20: The parameters utilization, affinity and hops-volume can be selected if they are considered on the cost function of the CB dynamic mapping algorithm.

4.4 Comparison of the Dynamic Mapping Algorithms

Tab. 4.1 presents the metrics used by the cost functions of the dynamic mapping algorithms introduced on this work. FF is for sure the fastest algorithm, since it requires only to find the next compatible IP core for a task. NN is also fast and tries to put the communicating tasks near to each other. BN comes next in terms of speed since it searches for possible IP cores according to NN and only uses PL when more than one candidate IP core is found. All the other algorithms consider all IP cores for making a mapping decision, therefore, they become slower with the increase of the number of IP cores. On the other hand, other algorithms can consider the channels of the NoC and the communication load of the tasks for preventing congestions. The computation load of the tasks mapped on an IP core is also an important metric for avoiding the overload of the IP core, and is considered by the CB algorithm.

Tab. 4.1: Metrics used by the cost functions of different dynamic mapping algorithms used on this work.

	Network position	Channel load	Comm. load	Task affinity	IP core utilization
FF	✓				
NN	✓				
MMC	✓	✓			
MAC	✓	✓			
PL	✓	✓			
BN	✓	✓			
MDE	✓		✓		
CB	✓		✓	✓	✓

In order to evaluate the different dynamic mapping algorithms, one synthetic application was developed and executed over a 4x4 and a 5x5 heterogeneous platforms based on the RENATO NoC. These heterogeneous platforms are configured with 2 DSPs, one on the upper left corner and another on the lower right corner. Both platforms reserve the IP core with address 22 for the mapper. All the other IP cores are GPPs. Fig. 4.21 illustrates the positioning of the mapper and GPPs on the 4x4 and 5x5 heterogeneous platforms.

DSP	13	23	33
02	12	Mapper	32
01	11	21	31
00	10	20	DSP

DSP	14	24	34	44
03	13	23	33	43
02	12	Mapper	32	42
01	11	21	31	41
00	10	20	30	DSP

Fig. 4.21: 4x4 and 5x5 NoCs representation of where the mapper and DSPs are placed on this case study.

The application is composed by 30 tasks, where 12 use the initial mapper and 18 are dynamically mapped. These 30 tasks are used and reused by a total of 15 sequencing actors, which describe different functionalities of the application. The computation time of the tasks range from 1,000 to 70,000 simulation cycles and the period of the tasks range from 200,000 to 500,000 simulation cycles. Each simulation cycle corresponds to one clock cycle of the real platform. The message sizes communicating the tasks range from 1,000 to 50,000 bytes. Six tasks are compatible to the 2 DSPs available on the platform, where these tasks have an affinity of 100% to them, while they have an affinity of 20% to the GPPs. Each platform-mapper combination was simulated for 10,000,000 simulation cycles and every time a task is mapped to a certain IP core it remains there until the end of the simulation.

Fig. 4.22 presents the execution time of 5 sequencing actors of the application for the 4x4 platform (graphs A-E) and the 5x5 platform (graphs F-J). The numbers 1 and 2 used on the label of the graphs indicate respectively the use of the CL and FF initial mappers. The graphs aligned on the same row indicate the same sequencing actor. Each graph presents on the X-axis the dynamic mapping algorithm used by the application. The Y-axis refers to the execution time of a sequencing actor in the form of a box plot, which is measured on simulation cycles. The execution time of a sequencing actor is the time between requesting the execution of the first task of the sequencing actor and the time when the last task of the sequencing actor has finished to execute.

On the first glance to any of the graphs, it is possible to see that the worse execution time of a sequencing actor can be more than the double of the best execution time, indicating that the mapping algorithm really influences the execution time of the application. One expectation was that the dynamic mapping algorithm CB would always present better timing results, since it considers both communication and computation of the application. However, CB provided the best timing results on only 55% of the cases presented on Fig. 4.22. While it is enough to know that the CB was the dynamic mapping algorithm that performed better on most of the times, it cannot be forgotten that the CB is also the one that costs more in terms of communication (i.e. information about communication load, task affinity and IP core utilization, which are information that need to be transmitted through the network from all IP cores to the

mapper). This costs in terms of time and network usage for transferring this mapping information is currently not considered on this work.

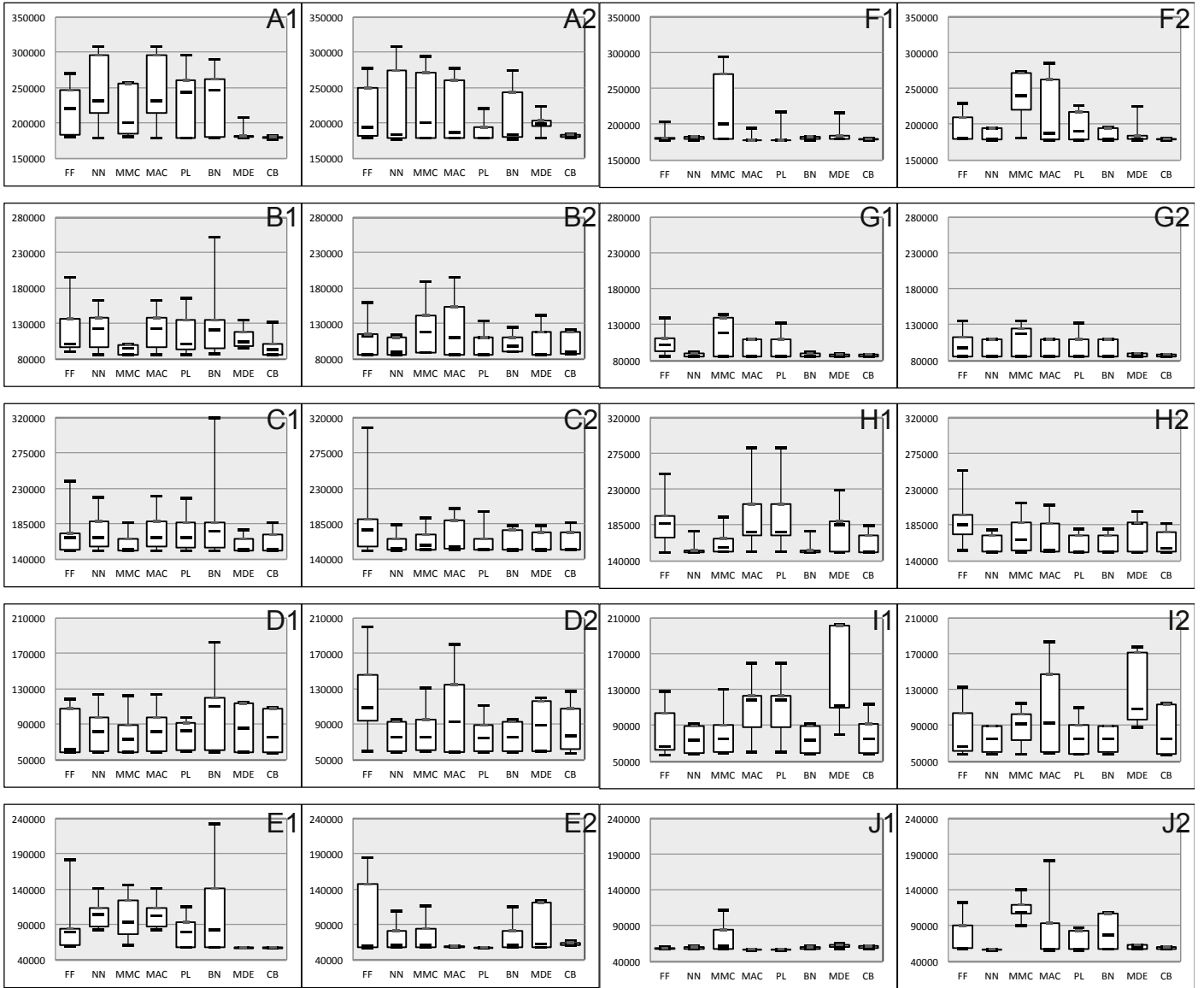


Fig. 4.22 Execution time of 5 sequencing actors of the application for the 4x4 platform (A-E) and the 5x5 platform (F-J). Graphs labeled with 1 use the CL initial mapper and graphs labeled with 2 use the FF initial mapper. The X-axis presents the dynamic mapping algorithm and the Y-axis presents the execution time of the sequencing actor following the box plot graphical analysis.

Another expectation was that the initial mapping algorithm CL would always present better timing results, since it creates clusters to keep the initial tasks of different sequencing actors apart from each other, thus giving space for the subsequent tasks mapped with the dynamic mapping algorithm to be grouped together with their corresponding initial task. However, the CL was only better or similar to FF on 42.5% of the times over the 4x4 platform and 82.5% of the times over the 5x5 platform.

Fig. 4.23 presents the total amount of congestion events and the average hops of the application for the 4x4 (labeled with 'A') and 5x5 (labeled with 'B') platforms with the initial mappers CL (labeled with '1') and FF (labeled with '2'). The average hops shows

how close the source tasks are mapped from the corresponding target tasks. If a target task is located in the same IP core as the source task, then the number of hops is 0. The greater the value, the more resources of the network is used to establish a data transmission. The amount of congestion events means the number of times that messages were temporarily blocked and could not be sent ahead towards its destination.

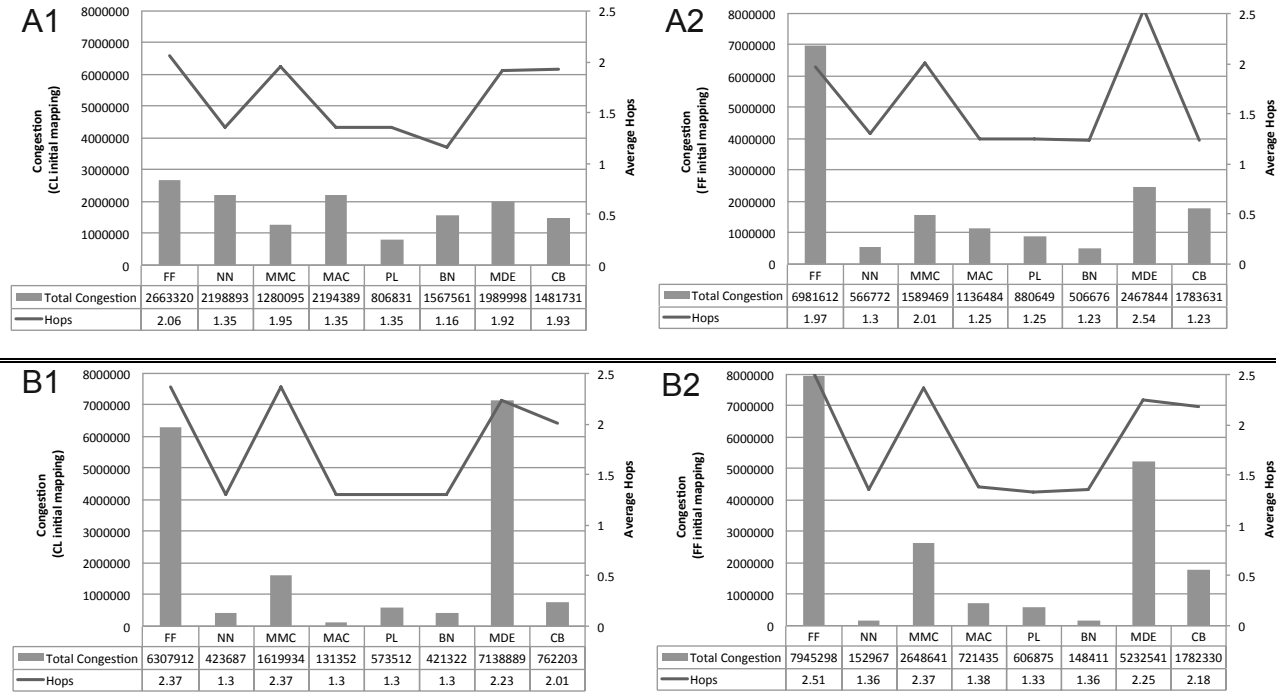


Fig. 4.23: Total amount of congestion events and average hops of the application for the 4x4 (labeled with 'A') and 5x5 (labeled with 'B') platforms with the initial mappers CL (labeled with '1') and FF (labeled with '2').

In a first glance to the four graphs of Fig. 4.23 it is possible to see that the dynamic mapping algorithms FF, MMC and MDE are usually the ones that present more congestion and number of hops. This indicates that greater communication distances have more chance to generate congestions. It is surprising to see that the CB is not among the dynamic mapping algorithms that had less congestion and lower average hops, since it was the one that performed better with regard to the execution time results presented on Fig. 4.22. When the 4x4 systems (graphs A) are compared to the 5x5 systems (graphs B), it can be seen that the dynamic mapping algorithms that performed well (NN, MAC, PL and BN) have improved on a bigger network, while the other dynamic mapping algorithms got worse. When the systems that use the CL initial mappers (graphs 1) are compared to the systems that use the FF initial mappers (graphs 2), it can be seen that most of the dynamic mapping algorithms have improved in terms of average hops and congestion events.

4.5 Task Migration

On the previous Section different initial and dynamic mapping algorithms were presented. Their goal is to map tasks for the first time when the tasks are not yet

mapped on the platform. The goal of task migration is to change the location of an already mapped task to a different IP core for improving the performance of the system and matching the dynamic behavior of the applications. So, the task migration service needs to be scheduled to execute periodically to check if all tasks are still mapped to an optimum place. The next Sections present and compare five different task migration algorithms.

4.5.1 Communication Task Graph Migration

The availability of complete communication task graph (CTG) at the design time can be of useful advantage for run-time management of tasks. It can significantly improve mapping assignment to minimize the unwanted delays, which are there due to a bad assignment of tasks or due to a change on the application. The objective of this migration algorithm is to try to remap currently mapped tasks that have bigger communication loads, closer to each other.

The detailed algorithm behind this CTG migration is as follows. Every time this algorithm runs, it goes through all mapped tasks. For each of these mapped tasks, it asks for its partner tasks (tasks with which it exchanges messages) and goes through each of them that are already mapped and are not in the neighboring IP cores of the requesting task. Then it picks up each of these partners, starting from tasks with bigger communication rates according to Eq. 4.1, and finds out whether that partner communicates with some other task with a greater communication rate. If not, the partner is checked whether this task was not yet requested by some other task for remapping. If yes, then again their communication rates are compared, and the greater one prevails. Thus, a candidate task for remapping was found, it is now required to find a new IP core for it. In this process, firstly an IP core is searched in the neighborhood of both requesting task and the task to be remapped. In this way, the cost of remapping is minimized to a minimum amount of hops. If no neighboring IP core is found, then the new IP core with least hop distance from the already mapped location is searched. If an IP core is found successfully, after checking its compatibility of type system and amount of tasks already mapped to the IP core, remapping is executed if the task is not active in the network. If the task is active (either sending or receiving a message), remapping can either be executed after completion of the message transfer or in the next firing cycle of the CTG migration algorithm. This whole procedure is repeated for each mapped task.

4.5.2 Runtime Communication Task Graph Migration

If the communication task graph (CTG) of the application is unavailable at the design time, it can still be extracted at runtime by probing each message that is being transferred through the network. The basic idea behind this runtime CTG migration is to continuously evolve the CTG, by collecting information (partner tasks and their mutual communication rates) from each message sent onto the network, and use this collected runtime CTG as the basis of migration, similarly as explained for CTG migration algorithm presented on Section 4.5.1. This algorithm clearly has as a

drawback a severe power and memory consumption, since data structures need to be maintained and updated for every communication performed on the system.

4.5.3 HotSpotScope Migration

As presented on Section 2.3, the platform model used on this work contains scopes to monitor the status the system. The main advantage of having these scopes is to use them for optimizing the application/platform, such as using them on a task migration algorithm. One of these scopes is the HotSpotScope presented on Section 2.3.5. This scope shows the number of negative acknowledgements that each port of each router of the network receives every time a message tries to be sent ahead on the network and it is not allowed due to congestion. With this kind of information in mind, the HotSpotScope migration algorithm tries to migrate the tasks responsible to generate more congestion.

The detailed algorithm behind the HotSpotScope migration is as follows. For each router on the network, its highest congested ports according the HotSpotScope are collected. If the amount of hotspots is greater than the previous cycle and greater than a minimum threshold, then all source and target pairs of communication passing through this congested port are considered for migration. The tasks considered for migration are then sorted according to hops distance against their communicating partner. Sequentially, a compatible IP core that can receive the task is searched at 1-hop distance from the partner task. As soon as the first candidate IP core is found, the task migrates if it is not active and the task migration algorithm finishes. If the task was active, the migration will happen as soon as the task becomes inactive.

4.5.4 Efficiency Migration

As all dynamic mapping algorithms except the CB do not use the affinity parameter on their cost functions, the efficiency migration algorithm tries to migrate the tasks to the IP cores where they have higher affinity. The detailed algorithm behind the efficiency migration is as follows. Every time the algorithm executes, all mapped tasks are considered for remapping to their best IP core in terms of affinity, if they are not already mapped to their respective best one. If there is more than one such best IP cores, the one that is located at least hops distance from the current IP core is selected, to minimize the cost of migration.

4.5.5 Surplus Migration

The idea of this Surplus Migration has been taken from Practical Fair-Share Scheduler (PFS) which is a fair-share process scheduler designed to support real-time workloads with soft (i.e., elastic) timeliness requirements [60]. The basic goal of this surplus migration, like any other fair-share schedulers is to maintain balance workload. In this algorithm, if a task accumulates a surplus greater than the threshold, the highest utilization non-running task is migrated towards the IP core containing the task breaching the surplus threshold. Intuitively, if a task has relatively high utilization and

is not running, it would likely be placed on another IP core if the migration costs were not an issue. Also, a task which has a high utilization is more likely to reduce the imbalance in workload that caused the large surplus to occur [60].

The detailed algorithm behind the Surplus migration is as follows. First of all migration destinations are found out using surplus metric defined in [61]. According to the authors, the surplus is the extra service received by a task compared to generalized multiprocessor sharing (GMS). They have provided an alternate estimation of GMS to calculate surplus of a task without the requirement of actual GMS simulation. To reduce the migration frequency, there is a threshold for surplus to exceed. For each of the migration destination, a candidate of migration is selected and migrated only if the final usage of migration the destination is less than the current usage of the candidate IP core.

4.5.6 Results

The same application presented on Section 4.4 was reused here to compare different task migration algorithms. Here, the CB was fixed as the dynamic mapping algorithm, since it was the CB dynamic mapping algorithm that provided the best results on Section 4.4. Fig. 4.24 presents the execution time of 5 sequencing actors of the application for the 4x4 platform (graphs A-E) and the 5x5 platform (graphs F-J). The numbers 1 and 2 used on the label of the graphs indicate respectively the use of the CL and FF initial mapping algorithms. The graphs aligned on the same row indicate the same sequencing actor. Each graph presents on the X-axis the task migration algorithm used by the application. The Y-axis refers to the execution time of a sequencing actor in the form of a box plot, which is measured on simulation cycles. The execution time of a sequencing actor is the time between requesting the execution of the first task of the sequencing actor and the time when the last task of the sequencing actor has finished to execute.

One expectation with regard to task migration was that it would be too costly to pay the price in terms of data to be transferred and time to justify migrating tasks at runtime. As it can be seen in all of the 20 graphs of Fig. 4.24, there was no case where any of the task migration algorithms were better than having no migration at all (the “no migration” result is presented as the first value on the X-axis of each graph). Besides, it is not all applications that can benefit from task migration. Applications need to be executing for a long time and need to perform representative changes at runtime in terms of computation, communication and associated tasks to justify the costs of task migration.

4.6 Summary

The inclusion of type system, application constraints, multi-tasking, task mapping strategies and migration algorithms to a system that can jointly execute the application, the platform, the mapper and the scopes bring a huge number of alternatives for creating heterogeneous MPSoCs and applications for them. Additionally, each part of the system can be further extended and parameterized, providing a resourceful tool for fine-tuning future multi-core platforms. On this context, task mapping strategies play

the most critical role on these systems, and bad mapping algorithms can result in significant degradation on their performance.

In this Chapter eight dynamic mapping algorithms for heterogeneous MPSoCs were compared on two platform configurations. These dynamic mapping algorithms consider different cost functions like the position of the task on the platform, the congestions of the network, the amount of data transmitted by the tasks, the affinity of the tasks to the different types of IP cores available on the platform and the utilization of the IP cores. Hence, the timing characteristics of computation and communication of the application and platform were taken into account on the presented case studies.

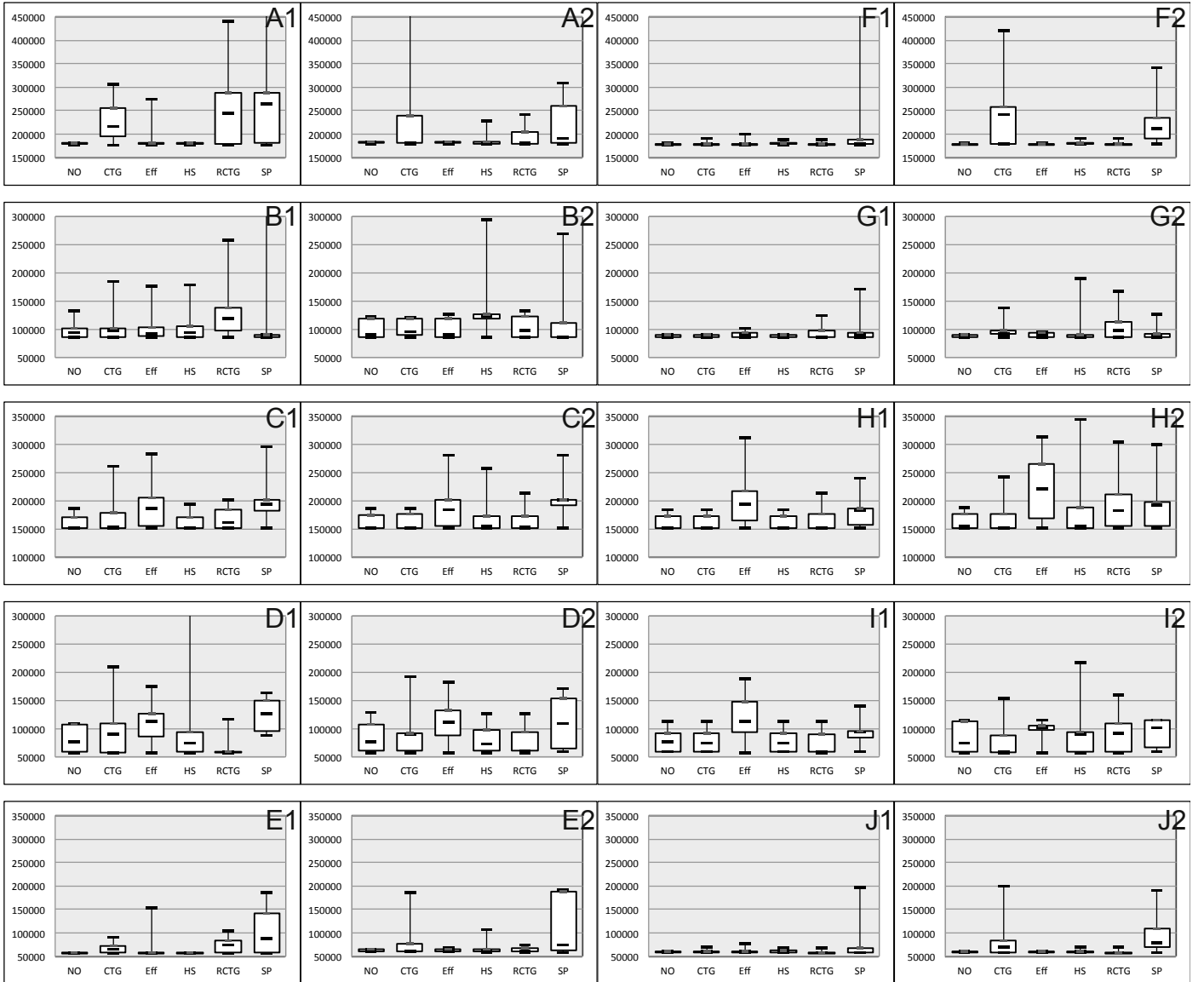


Fig. 4.24: Execution time of 5 sequencing actors of the application for the 4x4 platform (A-E) and the 5x5 platform (F-J). Graphs labeled with 1 use the CL initial mapper and graphs labeled with 2 use the FF initial mapper. The X-axis presents the task migration algorithm and the Y-axis presents the execution time of the sequencing actor following the box plot graphical analysis.

From the results obtained it was possible to see that in most of the cases the variance with regard to the timing to execute the sequencing actors are enormous. This is mainly

due to the fact that the RENATO NoC is employed, which does not contain any type of Quality-of-Service (QoS) like using a circuit-switched NoC or a NoC with priorities and virtual channels. The results also showed that the initial mapper CL was better than the FF on 42.5% of the times over the 4x4 platform and 82.5% of the times over the 5x5 platform. The dynamic mapping algorithm CB provided the best timing results on 55% of the cases.

Five different task migration algorithms were also presented. However, all the algorithms presented a poor performance partially because the application was short and did not present many dynamic changes in terms of computation and communication, and partially because the costs in terms of data to be transferred and time are too high to migrate tasks at runtime.

5 Reconfigurable Dual-Layer NoC[§]

While the previous Chapters focused on a model of a heterogeneous MPSoC and provided a big picture of it, they have also provided the main requirements for a successful system. Fig. 4.22 showed that even an application with only 30 tasks mapped to a platform that contains 25 multi-tasking IP cores can provide worse case execution times that are more than the double of the best case execution times of some parts of the application. This indicates that either some IP cores are too congested or the network is blocking due to congestion. While the congestion of the IP cores is a problem for the mapper algorithm to balance and was already tackled on Chapter 4, the congestion of the network can be reduced with a network with greater bandwidth.

However, increasing the bandwidth of the network requires increasing the number of wires between routers, the size of the buffers and the size of the crossbars. And these resources are already occupying too much area of the system while they are not improving the computation of the application. Therefore, this Chapter presents a novel way to increase the bandwidth of the network by creating a reconfigurable dual-layer NoC, while not replicating arbitration and routing algorithms and keeping the size of the buffers and crossbars low.

The proposed solution is based on the FPGA principle of sending a partial bitstream to define the interconnection of logic elements. Here the partial reconfiguration is used to change the routing table of circuit switching networks. In contrast to regular routers of a NoC, which use resource-costly multiplexers for switching, the FPGA interconnection system itself is used for circuit switching the paths between routers. Obviously, this adds some latency to the establishment of a path, but once a path is set, latency is expected to drop. Additional complexity is also expected to manage the partial reconfiguration process.

5.1 Dynamic Partial Reconfiguration

Dynamic Partial Reconfiguration (PR) refers to the capability of an FPGA to exchange a part of its configuration and replace it by a different one, while the rest of the FPGA keeps on running. This adds great flexibility and allows the reuse of resources, as components can be dynamically loaded when required, but do not waste resources when they are not. This approach is straight forward for an FPGA device which has a

[§] Major parts of this Chapter were presented at FPL [109].

configuration memory that determines its hardware functionality, as the memory can easily be changed. However, it also requires additional steps compared to a regular FPGA design to guarantee that the reconfiguration modules become interchangeable and the static part of the chip is not disturbed by the reconfiguration process. As of today, PR is basically a matter of research, while the quality of the required tools improves and the flow is simplified as the tools more and more automate the process. Nevertheless there are a lot of issues to be resolved before PR can be widely used in industry.

5.1.1 Introduction to Partial Reconfiguration

PR can be used to time-share hardware functionality of an FPGA. For this one or more PR regions (PRR) are defined. Each PRR can be configured using a corresponding PR module (PRM) in form of a partial bitstream. The area of the device which is not reconfigured at runtime is called the static region.

From an outside point of view PR has the same basic idea as the context switching of a microprocessor. Even though a simple microprocessor has only one arithmetic logic unit (ALU), it can execute multiple programs concurrently. For this it executes one process, respectively thread, at a time, and switches to the next one after a certain time or event. When this happens fast enough, it appears to the user as if all programs run in parallel. With this behavior, multiple processes can time-share one physical resource.

Using PR in an FPGA, the FPGA resources as FFs, LUTs and even more complex structures as RAMs and DSPs can be time-shared, together with the interconnection system. Fig. 5.1 illustrates the analogy between the microprocessor context switching and PR of an FPGA.

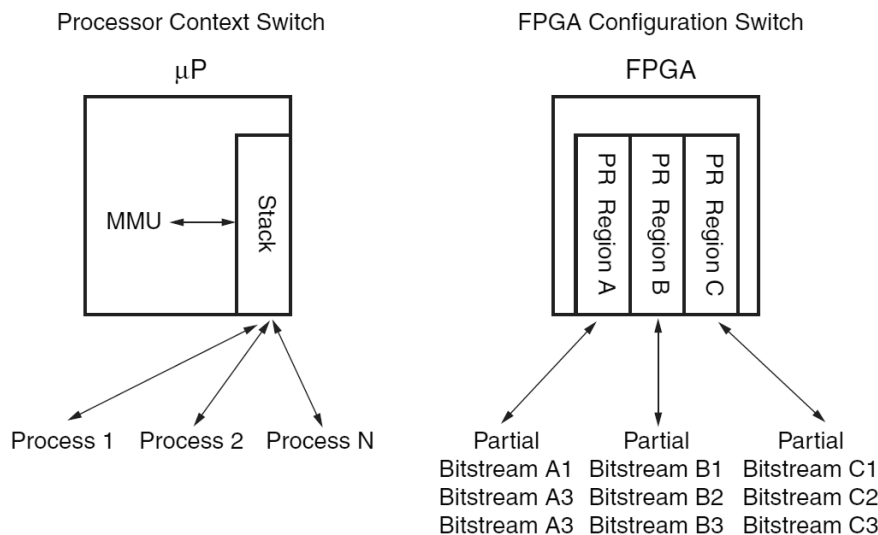


Fig. 5.1: Analogy between microprocessor context switching and FPGA partial reconfiguration [62].

5.1.2 The Early Access Partial Reconfiguration flow

The Early Access Partial Reconfiguration (EAPR) flow is a tool supported flow to

manage PR designs. It uses the classic ISE design environment and PlanAhead [62] together with a software overlay to support PR. The flow is described in detail in [62]. In addition to this, [63] describes the aid of PlanAhead as a software tool to visualize part of the reconfiguration design process and script the generation of the necessary files.

The EAPR flow adds strong restrictions to the design, especially with respect to hierarchy. The top-level of the hierarchy is used to black-box instantiate the static- and PR-modules, as well as global logic as DCMs, clock drivers and I/Os. In addition to this Bus-Macros (BM)s are required to connect the PRMs with the static logic. No other logic must be described at the top-level. Instead, it must be branched out and black-box instantiated as well. Fig. 5.2 shows the typical design of a top-level module satisfying the requirements for the EAPR flow.

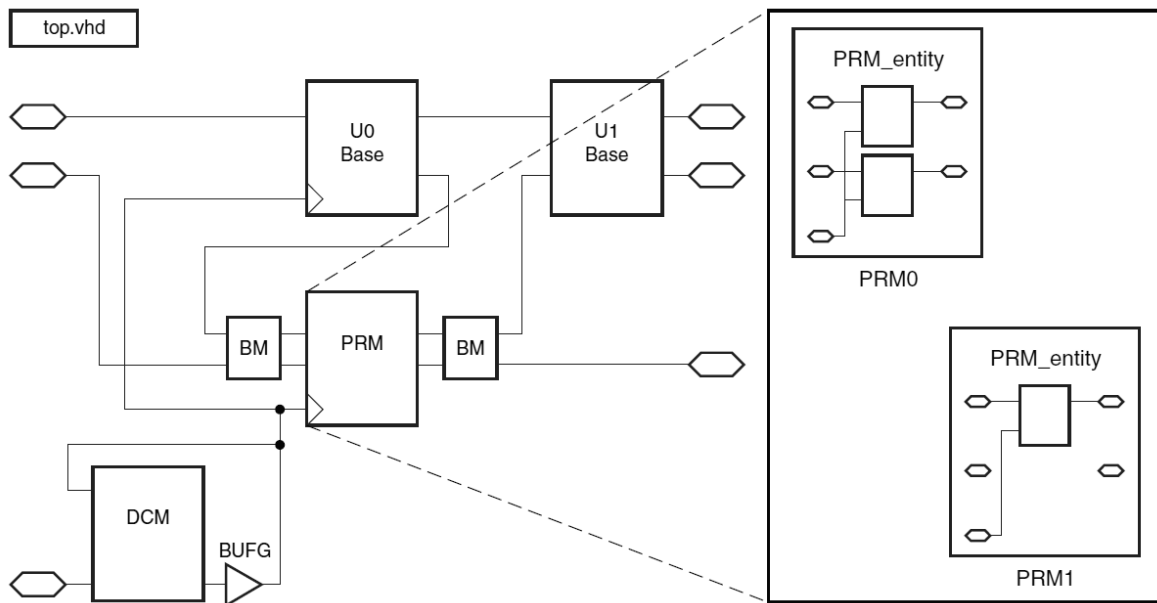


Fig. 5.2: Top-level module satisfying the EAPR flow requirements [62].

The BMs are pre-wired macros that overlap between the PR- and the static-region. The logic of the static region connects to the BMs on one side, and the logic of the PRR connects to the BMs on the other side, while the BM itself bridges the signals between these two. The BMs are used to guarantee that the static part of the design can successfully connect to the PR part, and that the individual PRMs are pin-compatible to each other. In case of PRMs having different pin-requirements, input pins can be left unconnected and output pins can be tied to an inactive state, but they cannot be omitted. The PRR itself may only use local signals passing through BMs and global clocks coming from BUFGs.

5.1.2.1 Flow description

Fig. 5.3 illustrates the EAPR design flow. The first four steps are similar to the flow of a regular design: the first step is to describe the design using a hardware description

language (HDL). For the PR design the strict separation of elements in the top-level module must be obeyed, as described in Section 5.1.2. The design should be extensively tested in simulation to ensure proper operation. The second step is to constrain the I/Os to meet the physical pins, and constrain the area groups and paths, typically the clock(s), to ensure the design meets the timing requirements. The third step is then to implement the PR design as a non-PR version, using one selected PRM for every PRR. In the fourth step the implemented design can be further analyzed for timing, and placement constraints can be set to ensure timing is met also in the implemented design. If necessary, steps two to four can be iterated until all constraints are met, and then the design can be finally tested on the target device.

Once the non-PR version of the design works as expected, the PR version can be implemented. For this, the static module(s) and the PR modules must be implemented separately, and then merged together. The result is an initial full bitstream, and a partial bitstream for every PRM. This process can be partly automated using PlanAhead, greatly simplifying and speeding up this phase.

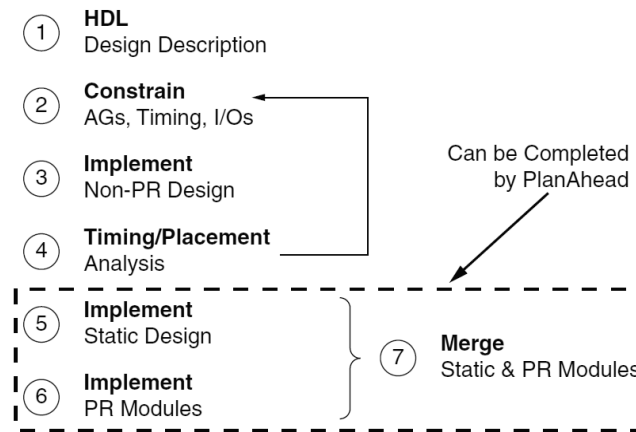


Fig. 5.3: EAPR design flow [62].

5.1.2.2 Defining Partial Reconfiguration Regions

To successfully place a PRR it is essential to understand the concept of configuration frames. Configuration frames represent the smallest parts of a device that can be (re)configured. For Virtex-II and Virtex-II Pro FPGAs, the configuration frames are formed by a full column of CLBs. Virtex-4 FPGAs allow the (re)configuration of smaller parts, as configuration frames align along the regional clock regions. They are one CLB in width and 16 CLBs in height. The Virtex-5 FPGAs are analogue to this, but have a height of 20 CLBs.

A PRR can be placed anywhere on the device, as long as CLBs are not split. However, placing a PRR along the configuration frame boundaries decreases the size of the partial bitstream files and speeds up the reconfiguration process, as all configuration frames that are straddled by the PRR must be completely stored and reconfigured. Also different PRRs may not share any configuration frame, as they would overwrite each other at the reconfiguration process.

To define PRRs PlanAhead can be used. It visualizes the area, showing the configuration frames and other logic elements that may or may not be added to the PRR. PlanAhead also generates the text-based AREA_GROUPS that describe the location and mode of the PRRs and adds them to the user-constraints files, simplifying the PR design process for the user.

5.1.2.3 Bus-Macro usage and background

BM is as introduced hard-wired connections between the static and the PR regions. The BMs basically consist of two CLBs: one in the static region, providing connectivity to the static parts, and one in the PRR, providing connectivity to the PRMs. These two CLBs are hard-wired together. When the location of each BM is locked, it allows the router to route all signals in the static region while ignoring the PRR, and later to route the signals within the PRR while ignoring the static region. The hard-wired signals guarantee the pin-compatibility of the individual modules. Fig. 5.4 shows a hard-wired BM which connects two CLBs with eight wires.

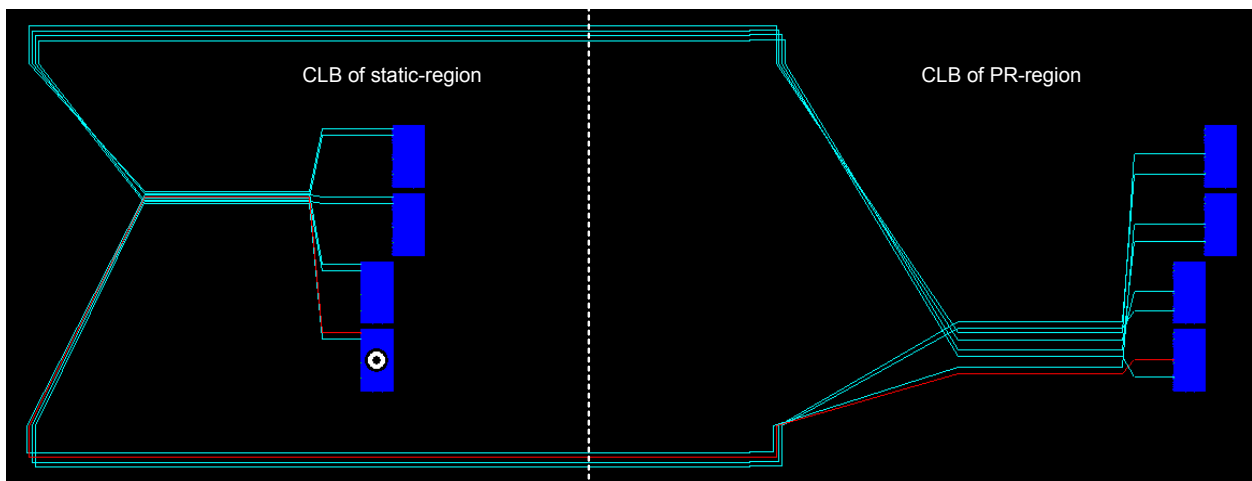


Fig. 5.4: Example of a Bus-Macro connecting two CLBs with eight wires.

Each FPGA-series requires its own BMs, as the FPGA's interconnection system is different. To allow higher flexibility, the BMs exist in different physical arrangements. Narrow BMs connect two CLBs next to each other, wide BMs connect CLBs that are separated by two other CLBs in-between. The latter version allows three BMs to be nested together, allowing 24 signals to pass through the boundary. This is illustrated in Fig. 5.5.

Except for the Virtex-5 BMs, the BMs are unidirectional, allowing the signals to pass in one direction only. They therefore exist in four different combinations, left-to-right (l2r), right-to-left (r2l), top-to-bottom (t2b) and bottom-to-top (b2t). Depending on the direction and the placement on the border, the inputs and outputs can be determined.

In addition to the physical properties, BMs can be asynchronous, letting the signals pass directly, or synchronous, using the FFs of the CLB to store the data for one clock cycle. The latter can enhance the timing of the design. Finally, BMs may have an enable-input

for each signal, allowing signals either to pass or tie the outputs to a defined state. This can be useful to avoid flickering of the signals while the PRR is reconfigured.

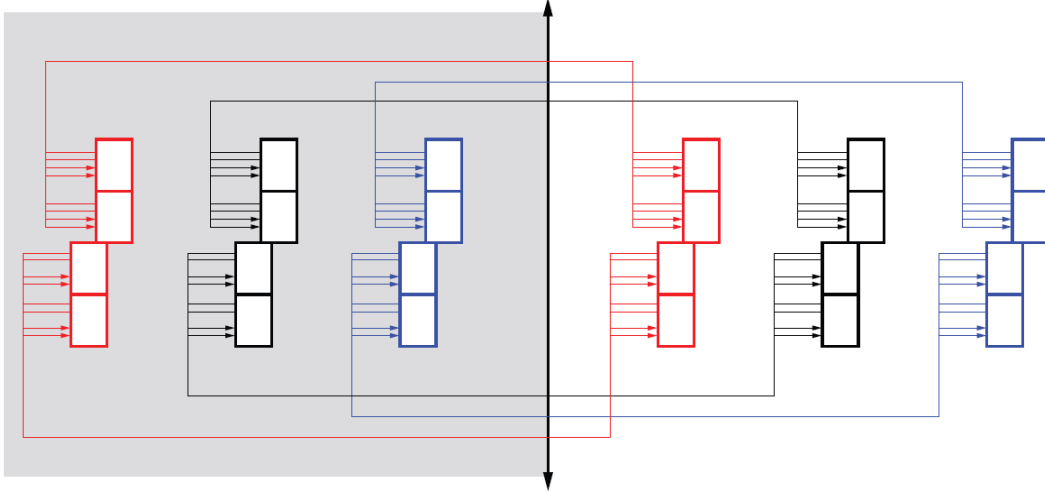


Fig. 5.5: Three nested Bus-Macros allowing 24 signals to pass the border between the static- and the PR-region [62].

5.1.2.4 Controlling the PR process

To control the PR process a PR-Master should be used. This PR-Master may act as follows. When one PRR is to be reconfigured, it first detaches the PRM by disabling the BMs connected to it, using the enable-inputs. Then a new partial bitstream is loaded into the device. This can be done internally using the internal configuration access port (ICAP), or externally using the JTAG interface. Then the new module may be reset using a classical reset signal, and finally the module is again attached to the static logic.

The PR-Master can be built in plain HDL, using a soft-core processor described in HDL, or using a build-in processor of the FPGA. The only (quite obvious) limitation is that the PR-Master cannot reconfigure itself.

5.2 Proposed Architecture

To use PR and control the PR process, a central PR-Master as described in Section 5.1.2.4 is needed. This PR-Master is supposed to receive switching requests, perform the PR accordingly, and then grant the route as established. Afterwards data can flow along the switched path.

Another important fact is that the NoC, as it was reconfigured to a specific path, now cannot be used to contact the PR-Master. Before doing so, a route to it would need to be established, which would need the work of the PR-Master itself, going round in circles. For this reason, a second NoC is introduced. This NoC is only responsible for the communication with the PR-Master, transferring small control messages only. To discriminate the two NoCs in this dual-layer NoC system, the first one will be called data network, and the second one will be called control network.

To keep changes from the IP cores and the NoCs as small as possible, and reuse the code

for route establishments and cancelations, a wrapper between the IP cores and the dual-layer NoC is introduced. The wrapper takes care about the communication with the PR-Master, and stalls traffic from the IP core to the data network until the path switching process is completed.

To test the described idea, a 3x3 NoC is designed, with the central data router being designed partially reconfigurable. Each router is connected to an IP core. Fig. 5.6 shows an overview of the system.

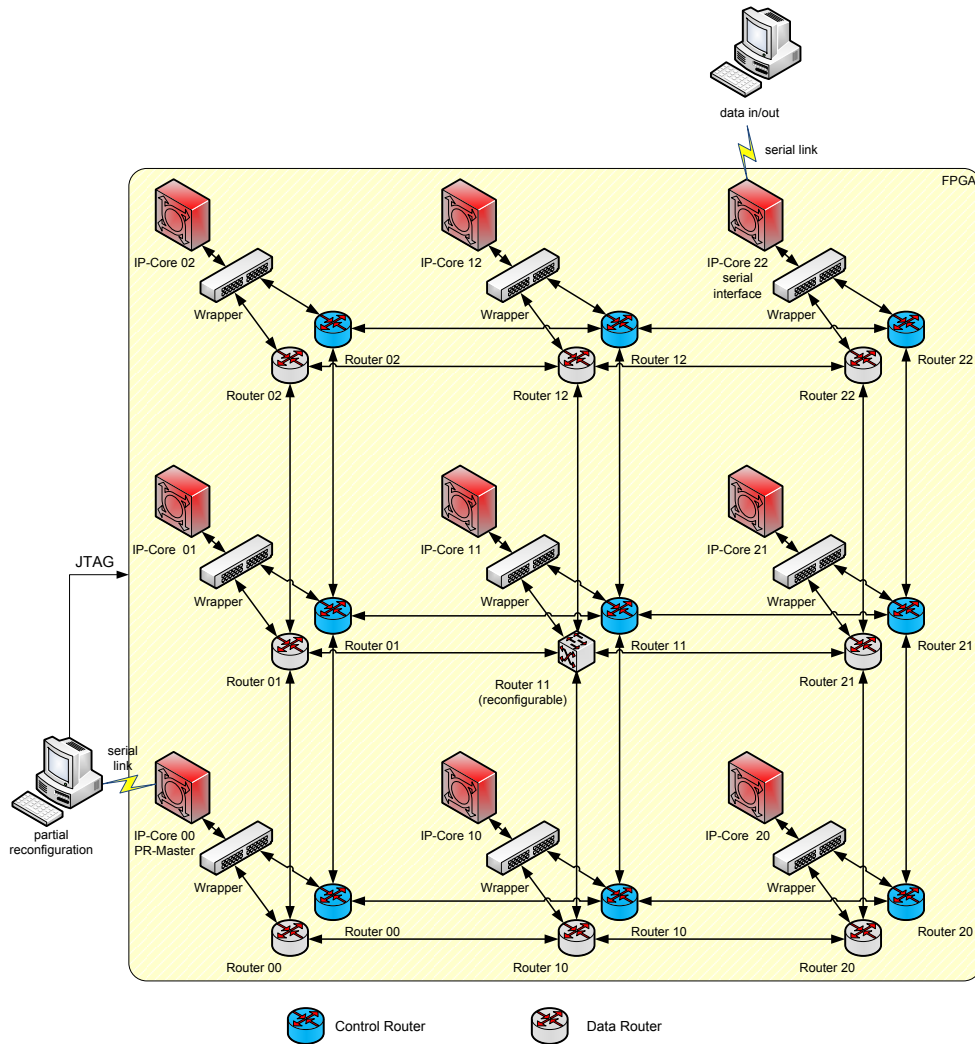


Fig. 5.6: Overview of the SoC showing the dual-layer NoC, IP cores, wrappers and serial links.

One IP core is set up as a RS-232 serial interface, receiving data packets from an attached PC and forwarding them into the data network. Another IP core is set up as the PR-Master, receiving route establishment and cancelation requests on the control network, and inducing the PR of the central data router. For this reason another serial link is connected to the PC, which receives the requests and reconfigures the FPGA using the JTAG interface. All other IP cores are set up as simple shift-and-forward modules, receiving a packet completely, modifying it, and forwarding it to the next IP core, using a predefined route stored in the packet.

5.3 Communication Protocol

Using a partially reconfigurable router makes it necessary to establish a route before a packet can be sent, and cancel the route after it is no longer used. For testing purposes the receiving IP core cancels the route just after completely receiving a packet. In a real system, the sender would notify the receiver when it has finished sending, and the receiver would then cancel the route. The complete process, including route establishment, data transfer and route cancelation, is shown exemplarily for a route from IP core 02 to IP core 10 in Fig. 5.6:

1. IP core 02 informs its local wrapper that it would like to establish a route to IP core 10.
2. The wrapper sends a route establishment request packet to IP core 00, which is the PR-Master. This control packet takes its way over the control network until reaching the destination (IP core 00).
3. The PR-Master is aware about the routing protocol, knowing that for this route the PR-Router 11 will need to connect its north input to its south output. It chooses a PRM that satisfies this need.
4. The PR-Master requests the chosen PRM by sending a request to a serially attached PC. The PC performs the PR using the FPGA JTAG interface, and confirms the successful PR to the PR-Master.
5. The PR-Master sends a confirmation packet back to the requesting wrapper (IP core 02). The packet travels again on the control network until reaching its destination (IP core 02).
6. The wrapper, which has so far stalled any traffic to IP core 10, now removes the barrier. The IP core itself does not need to be aware of the process, but can send out packets immediately after the route establishment request. The wrapper assures that no data will flow to the requested IP core before the route to it is established.
7. The data packets now can take their way to IP core 10, passing the PR router 11.
8. After the last packet of the communication has been received by IP core 10, it informs its local wrapper that the route can be canceled. This request must be send out by the destination IP core, respectively IP core 10, and not by the source, respectively IP core 02, as a packet might be in transit for some time and the route must not be canceled before the very last packet is successfully received.
9. The wrapper forms a cancelation packet, which takes its way to IP-Master using the control network.
10. The PR-Master removes the route from its internal table, which frees up the resources for later requests. No PR needs to take place at this time, and no confirmation packet needs to be sent out either.

As the resources of the PR-router are finite, the PR-Master may get a request that it cannot satisfy. In that case it denies the request by informing the calling wrapper. The wrapper is then supposed to send out a re-request after a certain time. This ensures that no packets get lost due to temporary impossible route request combinations, but will result in very poor performance. It should be avoided as far as possible.

5.4 Expected advantages and disadvantages

The described process of establishing and canceling a route is rather complex and consumes a certain amount of time. In addition to this, the active PR also consumes some time. For this reason, the process is expected not to be suitable for systems with very fast switching. However, many real systems are expected to have long-living connections between their communication partners. They could tolerate the time to establish a route, and make profit from the lower latency that is expected to be introduced by the circuit switching.

As this idea relies on the PR of an FPGA, it practically cannot be adopted to non-reconfigurable logic as regular ASICs. However, there are some ASICs that have user reconfigurable logic embedded to its fabric [64] and also examples of loosely coupled reconfigurable logic connected to ASICs [65][66]. FPGAs are still widely used in industry because of their low initial costs and the option to define their function at a very late point in the design steps, or even in the field

The PR-Master, wrappers and the second NoC for the control messages use additional resources of the FPGA. Compared to the data network, the control network does not need to be very fast and therefore may be designed much smaller. In contrast to this, the routers of the data network nearly vanish, as the interconnection system of the FPGA itself is used for switching. They may eventually be completely substituted by the interconnect system. This dramatically reduces the implementation costs of the data routers. Whether this outweighs the additional costs or not has to be examined. For a large system, the additional costs of the single PR-Master will carry no weight, and only the wrapper and control network will have to be taken into account.

5.5 Implementation

5.5.1 Choice of FPGA and software

To minimize the hardware development costs, an evaluation board already equipped with standard inputs and outputs can be used. Here the Xilinx ML403 board has been chosen, which uses the Virtex-4 FX12 FPGA and offers a serial interface and plenty of general purpose I/O ports [67]. Two of these I/O ports are connected to an external voltage level shifter IC that allows the use of a second serial interface.

The available software version for PR designs is the ISE development environment in version 9.2.4 with the PR software overlay in version 14. In addition to this, PlanAhead version 10.1 is used to graphically support the PR process and batch script the implementation of the different PRMs.

5.5.2 Hierarchy of the system

This Section describes the implementation of the project as illustrated in Fig. 5.6 and described on Section 5.2. The project is implemented in VHDL and follows the EAPR

flow guidelines for PR designs as described on Section 5.1.2. VHDL designs are in general hierarchical designs, encapsulating functionality in entities that may be reused at multiple points in the design. Fig. 5.7 shows the hierarchical design of the developed SoC that satisfied the restriction of the EAPR flow.

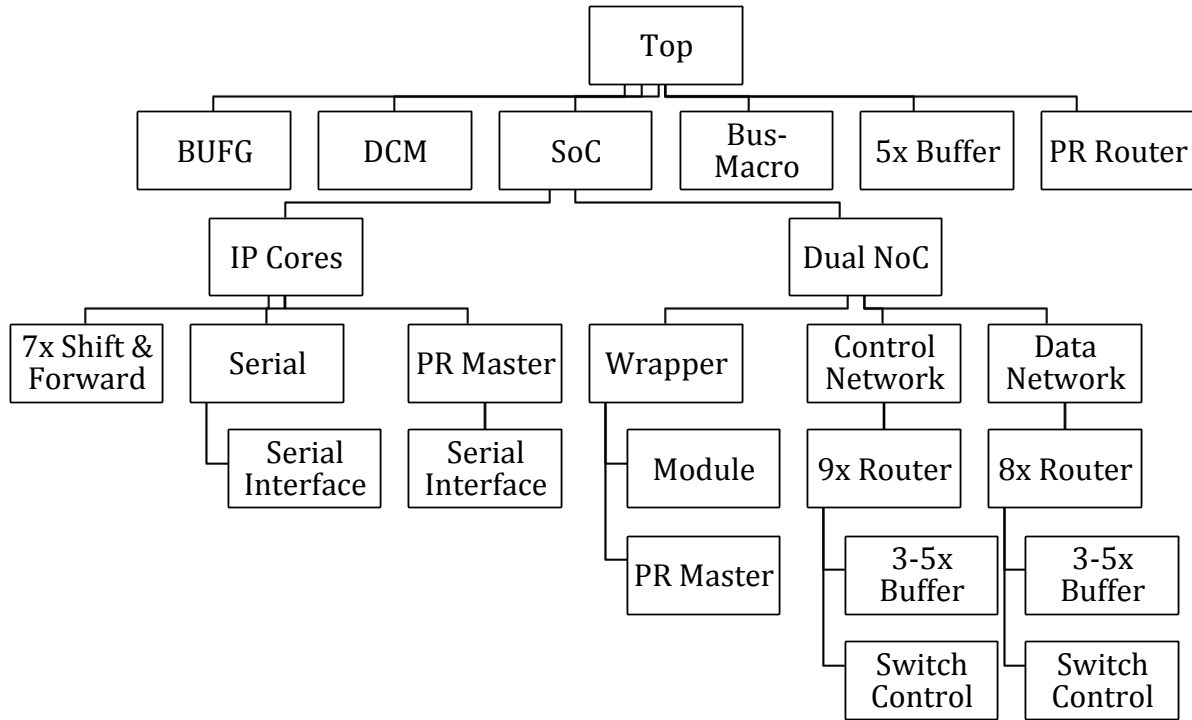


Fig. 5.7: Hierarchical design of the SoC, satisfying the requirements of the EAPR flow.

The top-level module may only instantiate different black-box modules and connect them with signals. No logic based on CLBs may be generated there. At first the top-level module instantiates a DCM that allows a flexible division and multiplication of the incoming clock signal. The generated clock is then fed into a clock buffer of type BUFG, making the clock available all over the chip.

Then the top-level module instantiates the SoC, which consists of the IP cores and the interconnecting dual-layer NoC. There are three types of IP cores in the system: a serial IP core that establishes a serial link to an external device, a Shift-And-Forward IP core that receives, modifies and forwards a packet, and a PR-Master IP core that initiates the PR-Process. The dual-layer NoC consists of the data and the control networks, each of which have nine logical routers, and a wrapper for each pair of data and control routers, which takes care about the establishment and cancelation of reserved routes as well as the assemble and disassemble of packet from both networks. The routers themselves have a switch control logic and three to five buffers, depending on their location on the network, as routers at the network border do not need to have buffers for directions without connections.

Fig. 5.7 also shows that the PR-router is not at its logical place together with the regular routers, as indicated by the dashed line. Instead, it is moved upwards in hierarchy to be black-box-instantiated from the top-module, as required by the EAPR flow. This is a

strong breach in hierarchy, requiring the signals connected to it to be passed through all modules up to the top-level module.

In addition to the main design, three options for the SoC are available: the first option is to have the central router be partially reconfigurable and have no own buffers; the second option is as the first but with dedicated buffers for the PR router; the third option is to have a regular router, allowing direct comparison of the PR designs with a regular design. To avoid plenty of changes to the system, the regular router is also connected at the top-level module. To choose one option, the `SYSTEM_VERSION` constant in the `SoCPackage.vhd` can be changed. This constant is evaluated by the top-level module and the wrapper module, instantiating the appropriate modules.

The individual modules will be discussed in greater detail in the following Sections.

5.5.3 Routers

The routers used on this work are the HERMES routers presented on Section 2.2.3. To describe the current switching mode of a router, the following convention is introduced. The inputs of the routers are per definition ordered as E, W, N, S and L (the four directions plus the local one). Then the output ports for every input port is named in the defined order. Fig. 5.8 illustrates an example of one possible switching mode of router and its associated routing table for this example. There the E input port is connected to the N output port, the W input port to the L output port and so on. The south input port is not connected to any output port. As the first line of the table is fixed per definition, the second one fully describes the switching mode, namely "NLW-E". This naming convention will be continuously used in the following Sections.

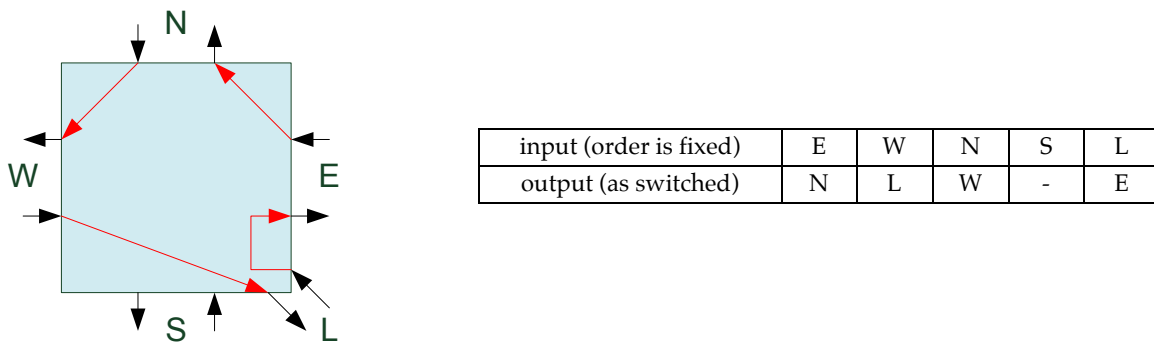


Fig. 5.8: Graphical illustration of one possible switching mode and the routing table of this configuration.

5.5.3.1 PR-Router

The PR-routers are pre-switched and hard-coded routers. In fact in their basic implementation they only consist of simple wires, connecting inputs to outputs. No logic that was previously needed for the regular routers are needed for them: the switching/multiplexing is realized by the interconnection system of the FPGA, the routing algorithm and arbitration is taken care about by the PR-Master. In the simple

version not even buffers or queues are needed. The PR-routers are described by a PRM that internally looks very much like Fig. 5.8. The only difference is that all signals are internally inverted. This is not actually needed by the design, but a requirement of the EAPR flow that does not allow signals to pass directly from inputs to outputs. The signals are then inverted again outside the PRM, resulting in the original signals again.

In general there are $5!=120$ possibilities to connect five input ports to five output ports with no one connected twice. For every of these a PR-router is generated and implemented as a separate PRM. As a couple of operations need to be done for every router individually, a program called SoC_RamGenerator was written in C# that automates these processes. At startup, it calculates the switching strings as described in Section 5.5.3 for all 120 routers. Then it allows to load a template PR-router and insert the strings, generating 120 files with one string each. In addition to this, it generates a batch-file that allows the batch synthesis of the files. The template file is designed to use a constant string in its headers to generate the appropriate connections. After the batch synthesis, 120 modules are available to be black-box instantiated at the top-level module and used as separate PRMs. Fig. 5.9 shows the developed program and some of the 120 automatically generated routers.

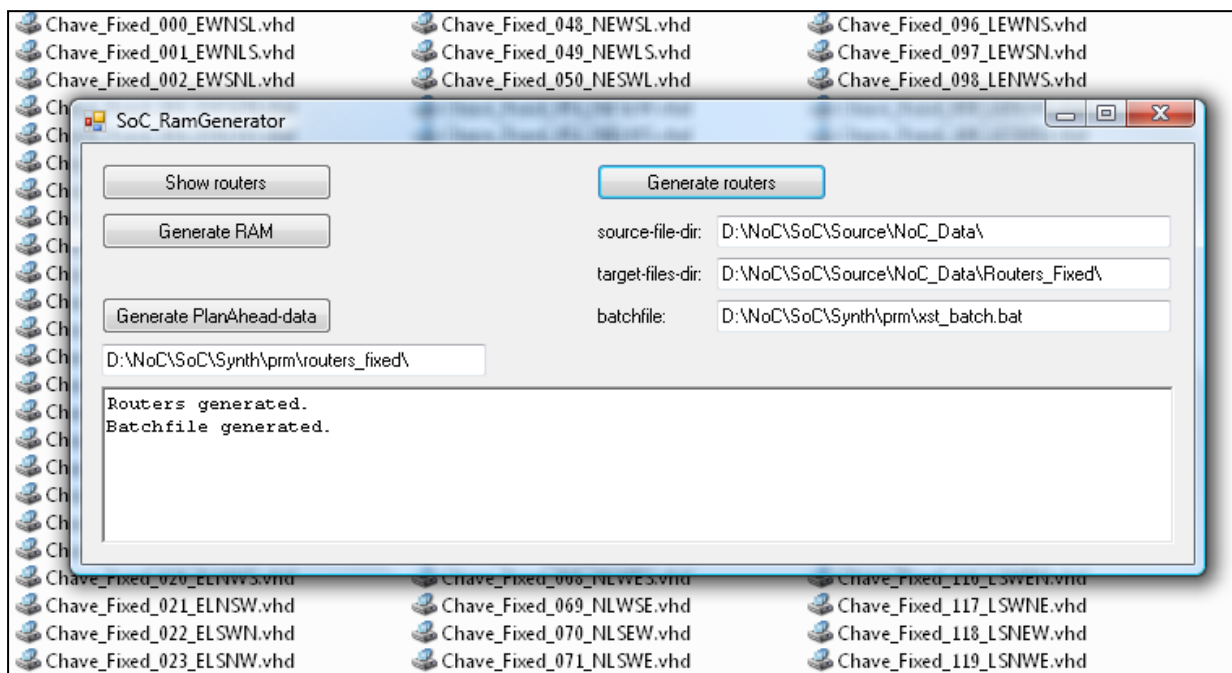


Fig. 5.9: The developed program "SoC-RamGenerator" that generates one pre-switched and hard-coded router for every of the 120 possible switchings.

5.5.3.2 PR-Router with dedicated buffers

The PR-routers with dedicated buffers are an enhanced version of the simple PR-routers described in Section 5.5.3.1. They use the very same PRMs, but additionally have buffers at their input ports. The buffers are stripped down versions of the buffers used for the regular routers. They are designed to store a single flit each, keeping them small and efficient. In contrast to the regular routers, no routing algorithm is executed that needs additional time. Therefore flits can be forwarded directly at the clock cycle followed by

the one where they have been received, making larger buffers unnecessary. The buffers are instantiated at the top-level and are logically positioned right before the BMs that build the entry point to the PRR.

5.5.3.3 Regular router

For comparison of the PR-design to a regular design a regular router can be inserted at the place of the PR-router. To keep the required changes to a minimum, the regular router is also instantiated at the top-level module, even though the absent of PRRs frees from the restrictions of the EAPR flow. The unnecessary signal forwarding to the top-level module will also have no effect to the final design and will not result in additional hardware costs.

5.6 System-on-Chip

The SoC consists of the IP cores and the dual-layer NoC, where both NoCs are arranged in a 3x3 mesh topology. The individual IP cores and the construction of the dual-layer NoC will be described in the following Sections.

5.6.1 IP cores

The current design consists of three different IP cores: the Serial Interface Core, the PR-Master Core and the Shift-And-Forward Core. The first two Cores are instantiated once each, whereas the Shift-And-Forward Core is instantiated seven times. All three Cores are based on a Mealy finite state machine (FSM). A finite state machine can be described by a flow diagram, where the machine changes its state when the conditions on the transitions are fulfilled. The described FSMs all work synchronous with regard to the central clock, with exception of the asynchronous reset signal. In contrast to a Moore FSM, where the output signals are a function of the current state only, a Mealy FSM computes its outputs from the current state and the input signals. The following three Sections will describe the individual IP cores with FSM state diagrams that are simplified to their core functions.

5.6.1.1 Serial Interface Core

The Serial Interface Core allows the SoC to establish a serial communication link to a different device, here a PC, using a serial RS232 interface. For this it uses a serial module developed by Fernando Gehm Moraes of the GAPH group [68]. The IP core acts as a bridge between the serial interface and the NoC.

Basically the IP core consists of two nearly independent parts. One part receives data from the serial interface and sends it towards the network (s2n), and the other part does vice versa (n2s). Both parts use the data network for the transmission of data, and the control network for the establishment and cancelation of routes. For the latter the attached wrapper is used. Fig. 5.10 shows the Serial Interface IP core as a simplified FSM. Its function will be then described in detail.

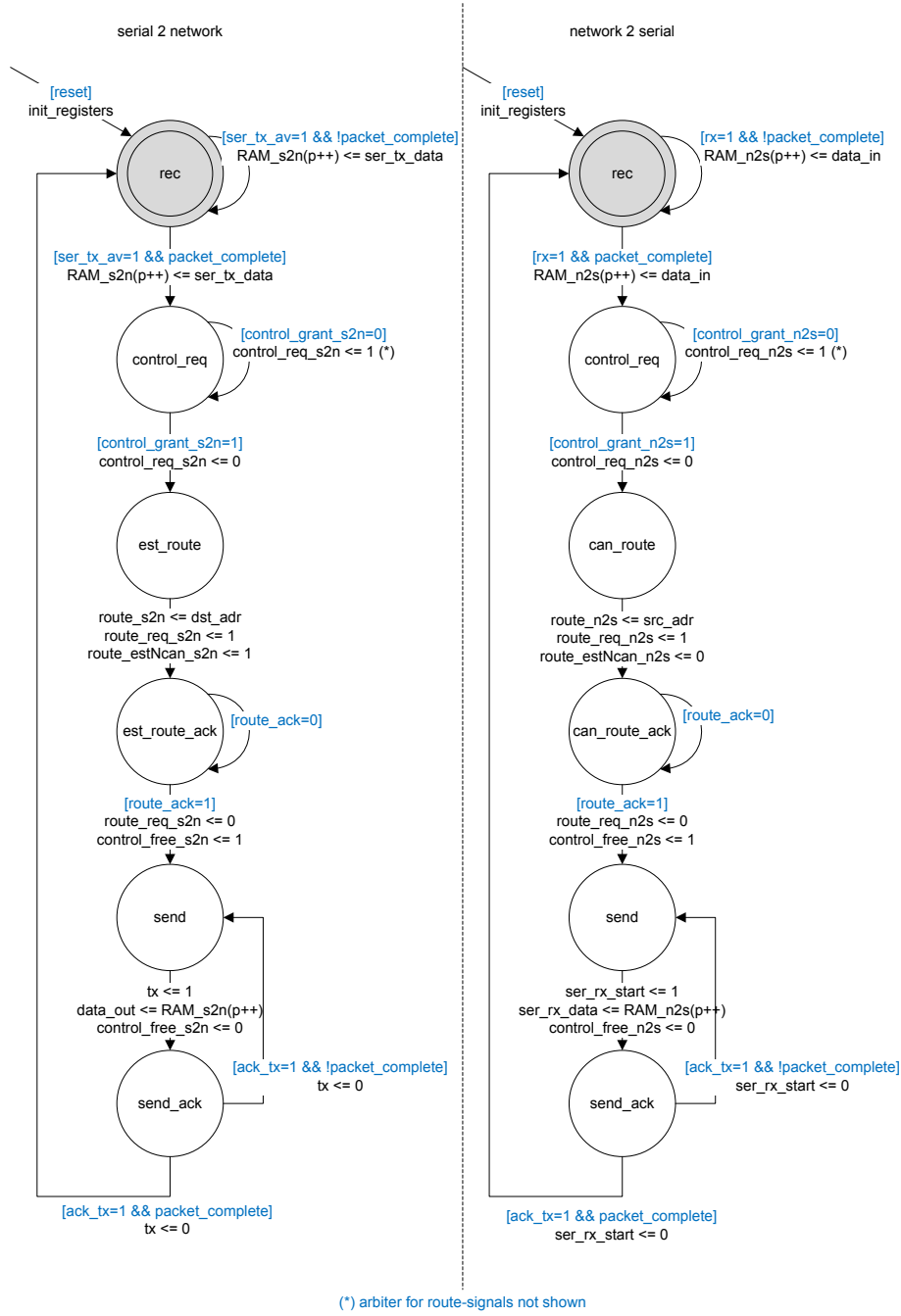


Fig. 5.10: Simplified finite state machine of the Serial Interface IP core.

The left part of the FSM describes the s2n communication. At first the FSM must be asynchronously reset; this will initialize its registers and safely start the FSM in the receive (*rec*) state. There it can take in data from the serial module and store it in its local RAM. Every time the *ser_tx_av* signal goes high, one flit is taken in from the *ser_tx_data* signals and stored in RAM, incrementing a pointer to the next free position in RAM. The FSM can determine whether the packet is completely received or not by analyzing the second flit, which represent the packet-size excluding the two-flit header. Once the packet size is known, it is stored internally in the *size_s2n* register and the *size_valid_s2n* signal is asserted high. Fig. 5.11 shows this first part in simulation.

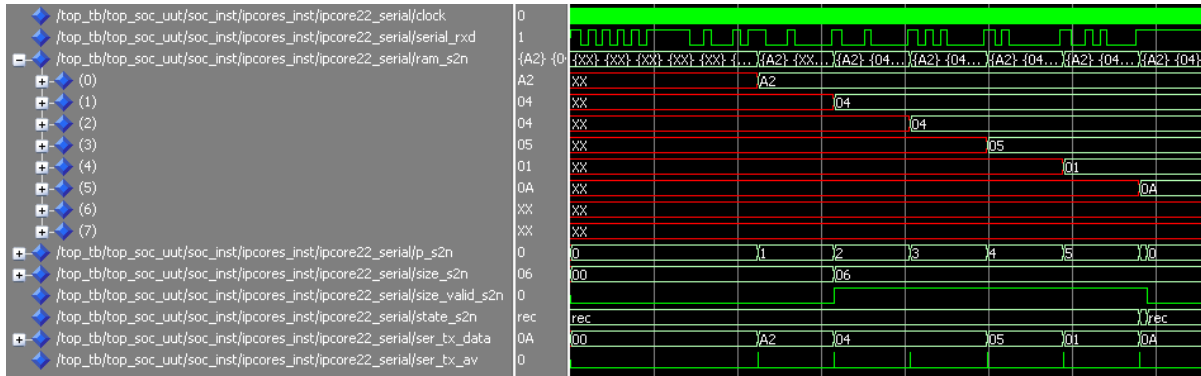


Fig. 5.11: Simulation of the Serial Interface IP core - receiving of a packet.

Once the packet is completely received and stored in RAM, the FSM goes into the *control_req* state. This is the only state that interacts with the second FSM, as both need access to the *route* signals connected to the attached wrapper. A central arbiter, which is not shown in the figure, grants access to one of the FSM at a time only, time-sharing this resource. After the arbiter grants the request, the FSM goes into the *est_route* state and requests the establishment of a route from itself to the destination of the packet. For this it raises the *route_req* signal and puts the destination address on the *route* signal. In addition to this, it sets the *route_estNcan* signal to one, indicating a route establishment. The FSM then waits for the request being acknowledged in the *est_route_ack* state. The attached wrapper, which receives this request, takes care about forming a route-request packet and sending it to the PR-Master. The FSM itself releases the *route* signals by setting the *control_free* signal to one and continues to the *send* state. Fig. 5.12 shows this in the simulation.

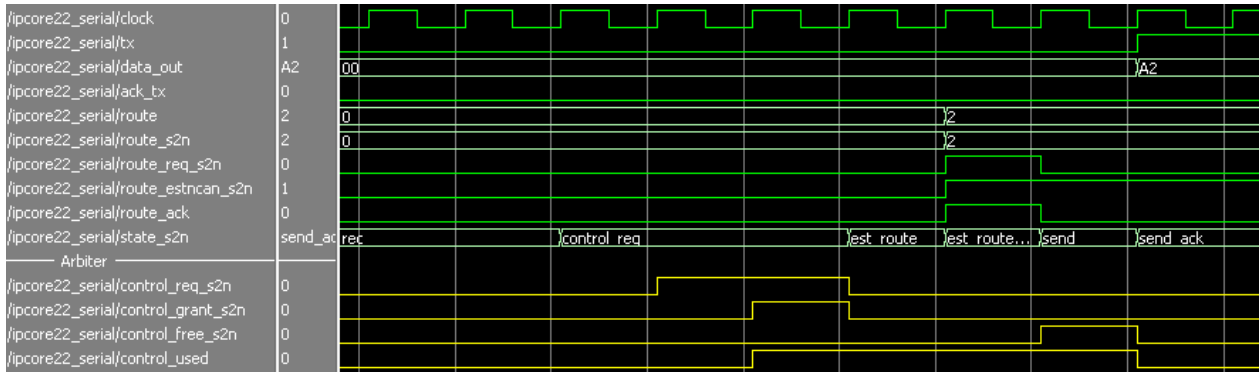


Fig. 5.12: Simulation of the Serial Interface IP core - route establishment request.

Finally the FSM sends out the packet flit by flit to the data network. For this it raises the *tx* signal and puts a data flit to the *data_out* signals. Then it goes to the *send_ack* state, waiting for acknowledgment from the data network. These two states loop until all flits of the packets are sent. Finally the FSM goes back to the *rec* state, waiting for a new packet from the serial module. Fig. 5.13 shows the sending of the packet. Important to notice is that the FSM does not need to wait for the route to be established. The wrapper will stall any traffic to a pending route not yet established, allowing the FSM to start transmitting immediately after the route request.

The n2s part of the IP core works very similar to the s2n part. The main difference is the

direction of the traffic flow, as the data there is received from the data network, and sent to the serial interface. In addition to this, the s2n part cancels a route instead of establishing one.

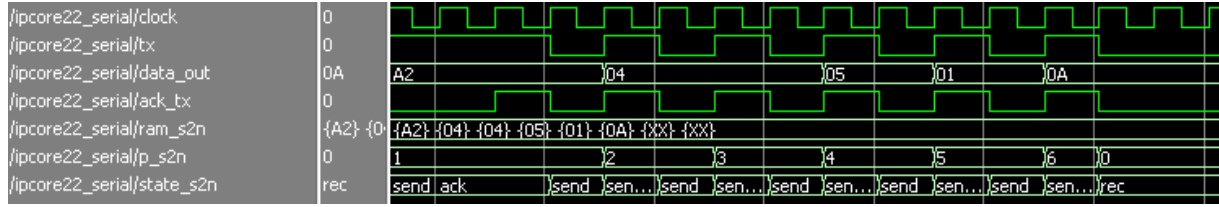


Fig. 5.13: Simulation of the Serial Interface IP core - sending of a packet.

5.6.1.2 Shift-And-Forward Core

The Shift-And-Forward (SAF) Core is a small IP core that receives (stores) a packet sent to it, modifies it, and sends it out (forward) again. The packet follows a pre-defined route stored in it. For this the IP core circularly shifts the packet content as depicted in Fig. 5.14 for an example packet of the size of five flits. The routers interpret the right half of the first flit as the destination address and the second flit as the size of the packet. The rest of the packet can hold user data. To simplify the process, starting with the third flit each flit holds the address of an IP core on its way. These addresses are circularly shifted left at every SAF-Core they arrive, letting the packet flow in the pre-defined route. The left half of the first flit is used to store the address of the source IP core. It is needed by the SAF-Core to cancel the old route. Afterwards it is replaced by its own address as the new source.

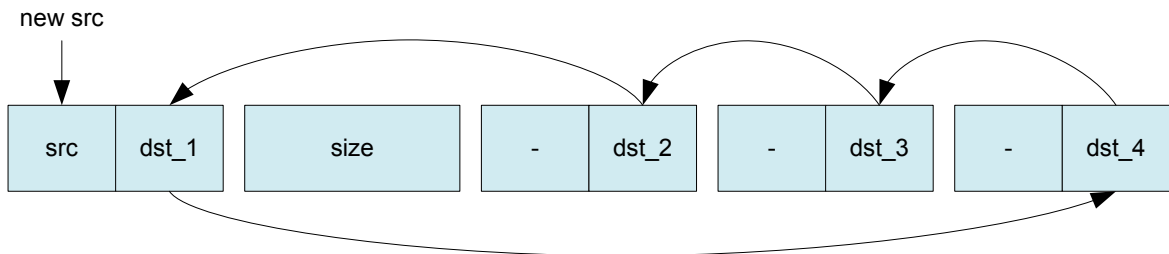


Fig. 5.14: Circular shifting of a packet to follow a pre-defined route.

Fig. 5.15 shows the simplified FSM of the SAF-Core, which works as follows. As the handshake signals work the same way as the ones of the Serial Interface IP core described in Section 5.6.1.1, only the main steps are described here. After the asynchronous reset the FSM starts in the *rec* state. There it receives a complete packet and goes to the *can_route* state, where it cancels the route that the packet came from. It determines this from the first flit of the packet, where the source and destination address is stored. After the acknowledgement of the wrapper is received in the *can_route_ack* state, the FSM goes to the *modify* state. There it loops through the packet, modifying it as described above. To save resources this is done at a speed of one flit per clock cycle. An alternative method is to do the modification in a single clock cycle, saving time but using much more FPGA resources. As this is a demonstration IP core only, the slow method is used, saving resources.

After the modification is complete, the FSM goes into the *est_route* state where a new route from the current IP core to the new destination IP core of the packet is established. After another acknowledgement of the wrapper in the *est_route_ack* state, the packet can be sent out. For this the FSM loops in the *send* and *send_ack* states, until the complete packet is sent. Then the FSM returns to the initial *rec* state. Fig. 5.16 shows the simulation of the receiving, the route cancelation, the packet modification and the route establishment.

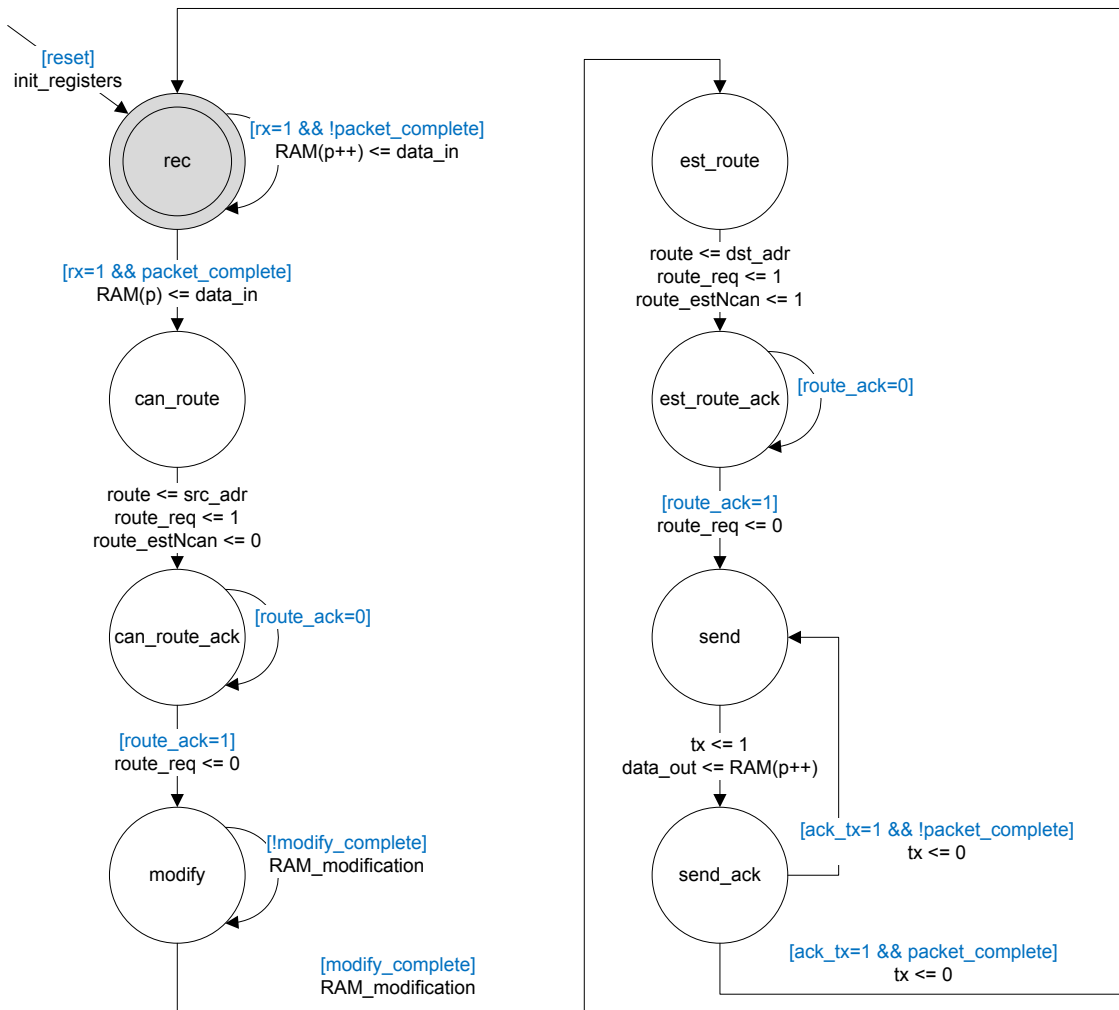


Fig. 5.15: Simplified finite state machine of the Shift-And-Forward IP core.

5.6.1.3 PR-Master Core

The PR-Master Core is the central place where route requests are received. The PR-Master then decides whether or not to partially reconfigure the FPGA. If yes, it induces the PR-process by sending a PR request to a serially attached PC. After the successful PR, the PR-Master confirms the establishment of the route to the sending IP core. If no

PR is necessary, the confirmation is sent out directly. In case of a route request that cannot be granted due to a lack of resources, the PR-Master denies the request.

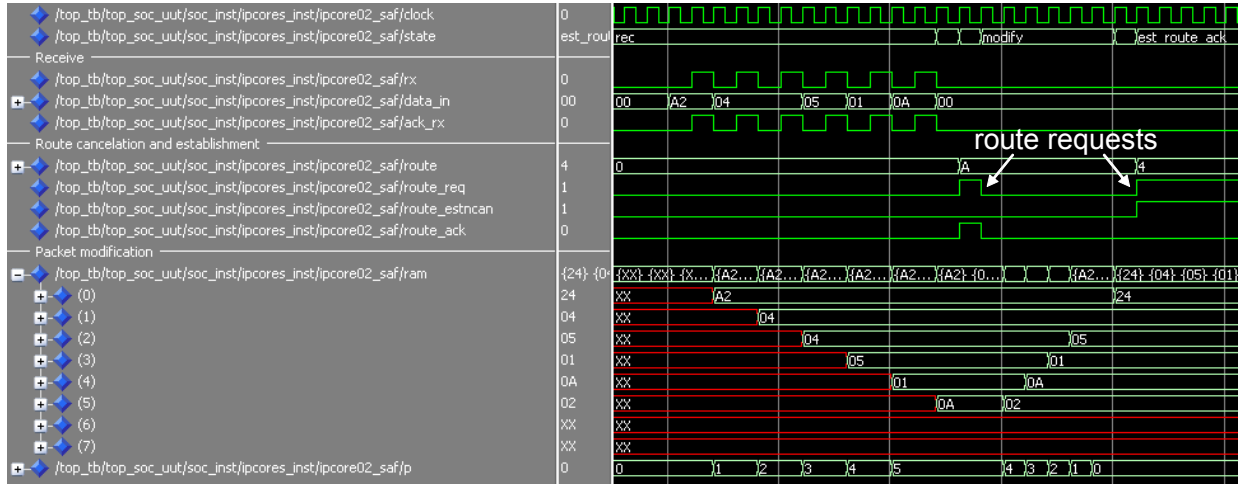


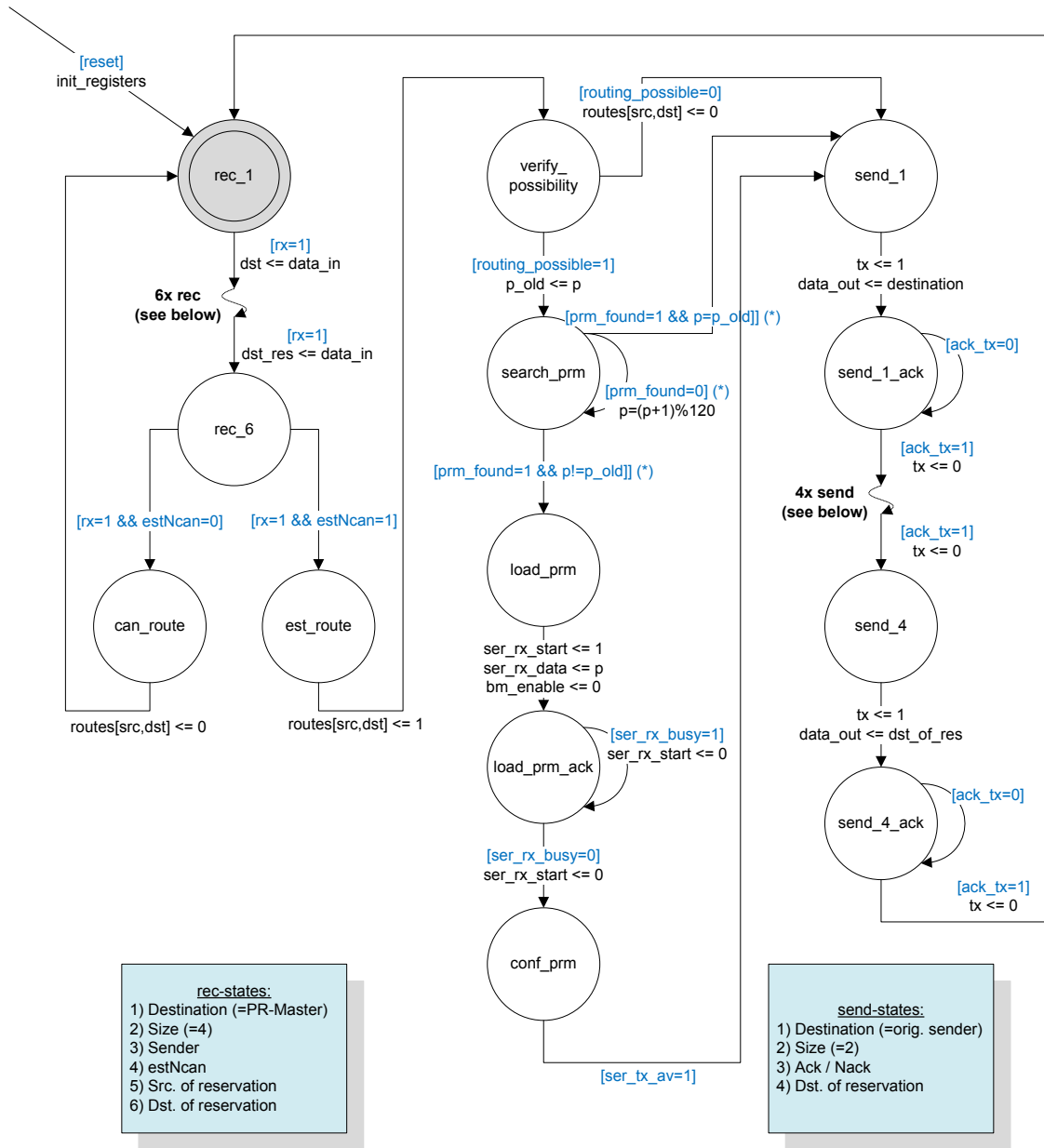
Fig. 5.16: Simulation of the Shift-And-Forward IP core.

The PR-Master is implemented as a regular IP core. This makes it easy for the other cores to communicate with it. Their wrappers, responsible for the establishment and cancelation of routes, only need to send a packet to the address of the PR-Master using the control network. However, in contrast to all other IP cores which are directly connected to the data network only, the PR-Master Core is supposed to have a connection to the control network instead. To keep the interfaces untouched and avoid a lot of changes of the connection system, the PR-Master gets a special wrapper. This wrapper connects the control network directly to the IP core. As the data-width of the data network is larger than the one of the control network, this special wrapper has to pad the signals with zeroes. With this change, the PR-Master can receive requests similar to the Serial Interface Core of Section 5.6.1.1, using the same handshake signals as described there. Fig. 5.17 shows the simplified FSM of the PR-Master and the structure of the request and response packets.

At first the FSM receives the complete request packet which consists of six flits as shown at the bottom left. The fourth flit determines whether a route is to be established or canceled. To keep track of all active routes, the IP-Master has an internal table called "routes". The table is of size nine-by-nine, having an entry for every route with the column indicating the source address and the row indicating the destination address.

When a route is canceled, the entry is simply removed from the table and the FSM returns directly to the *rec* state. No confirmation packet is needed to be sent, as this packet would be of no interest for the requesting IP core. When a route is established, it is entered into the routes table. Then the FSM goes into a verification state. There the routes table is translated into a table for the central PR router, describing which inputs need to be connected to which outputs. This table is the requesting table called *table_req*. As each router has five inputs and five outputs, this table is of size five-by-five. Each input can be connected to one output only, and each output can be connected to one input only. In other words, the *table_req* may have only one entry per row, and one entry

per column. Otherwise, one input or output would have to be connected twice, which is not allowed. The complete check is realized by pure combinatorial logic, and the result is stored in the signal *routing_possible*.



(*) *prm_found* is a complex function that compares the requesting "routes" table with RAM(p) that stores all possible routings per PRM

Fig. 5.17: Simplified finite state machine of the PR-Master IP core.

If routing is not possible, the entry from the routes table is removed and the FSM goes into the send state, denying the request. Otherwise, routing is possible. As described in Section 5.5.3.1 there is one PRM for every possible combination of inputs to outputs. Therefore for every *table_req*, which is possible to route, there is at least one PRM that satisfies this. To find one of them easily and efficiently, the routing-table of each PRM is

pre-computed and stored in a ROM (as the Virtex-4 has no ROM, a RAM with permanently deserted write signals is used). The FSM uses a pointer into the ROM, starting at the currently loaded PRM. If this already satisfies the *table_req*, by means of having an entry at every position the *table_req* has, no PR is required and the FSM continues to the send state, confirming the route request. Otherwise the FSM goes into the *load_prm* state. There it sends the number of the PRM to a serially attached PC, which then performs the PR task. After completion, the PC confirms this by replying to the serial link, and the FSM goes forward to the send state.

The send states consists of four sending states for each of the four flits to be sent and four corresponding states waiting for the acknowledgement of the network. The structure of the reply packet generated there is also described in Fig. 5.17. After sending the reply packet, which is either an *ack*, confirming successful establishment of the route, or a *nack*, denying the request, the FSM returns to the initial *rec* state.

Fig. 5.18 shows how the PR-Master searches for a suitable PRM. First it adds a request to the routes-table (1), showing a route from IP core 6 to IP core 1 (IP core 1 is represented by a binary 000000010_2 , counting starts from 0). This is then directly translated into the *table_req* table (2), showing that a connection from north (2=north) to south (3=south, therefore bit #3 set: $01000_2=8_{16}$) is needed. Then the FSM increments the pointer into the RAM and reads out a table. Once this table matches the need of *table_req* (3) the signal *prm_found* goes high (4). This is the case at p=2, the FSM therefore continues to load PRM number 2 (5).

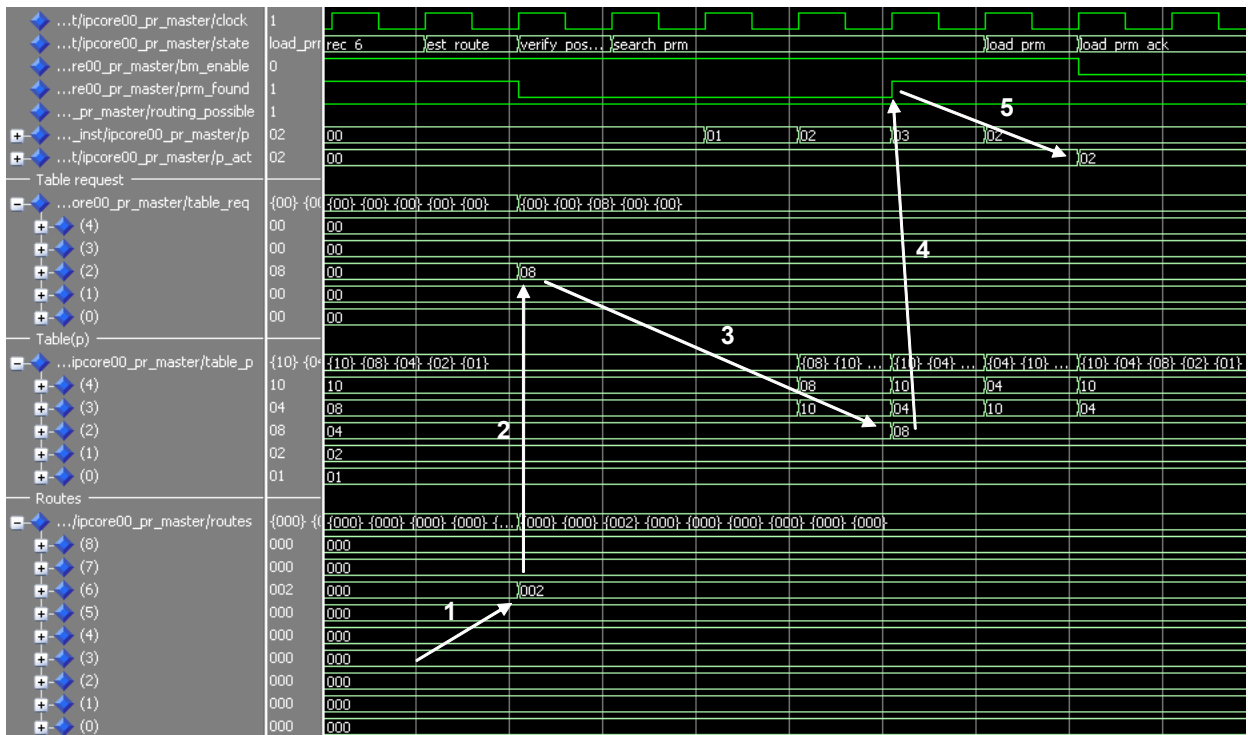


Fig. 5.18: Simulation of the PR-Master IP core - searching for a suitable PRM.

The content of the RAM is generated by the program SoC_RamGenerator written in C#. It has 120 entries, one for each of the PR-routers. Each entry has the size of 25 bits,

storing the five-by-five switching table. The RAM only supports data widths of 1, 2, 4, 9, 18 and 36 bits, but not the needed 25 bits. To reduce complexity, the 36 bit data width is used, padding the additional 11 bits with zeroes. The data to be stored then still fits in one single block-RAM. The SoC_RamGenerator shown in Fig. 5.19 computes the 120 entries, pads them accordingly and converts them to the hexadecimal representation that is typically used for the pre-defined content of a block-RAM in VHDL.

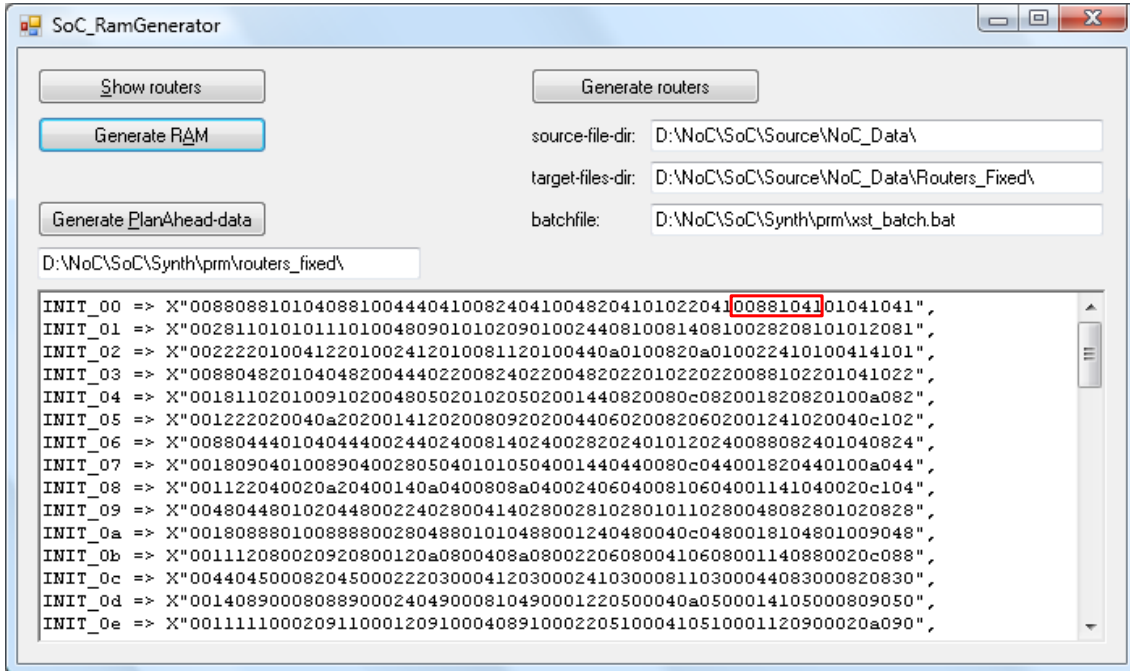


Fig. 5.19: Generation of the block-RAM content for the switching tables of the PR-Master.

As an example the second data block is marked in red. It describes the switching table for the second router with the switching "EWNLS". Fig. 5.20 visualizes the conversion from the hexadecimal string to a switching table. For this the hexadecimal representation is converted to binary, the padding zeroes at the left side are removed and the rest is rearranged to a table of size five-by-five. The table has its MSB at the lower right and the LSB at its upper left, making the binary string appear to be inverted in order.

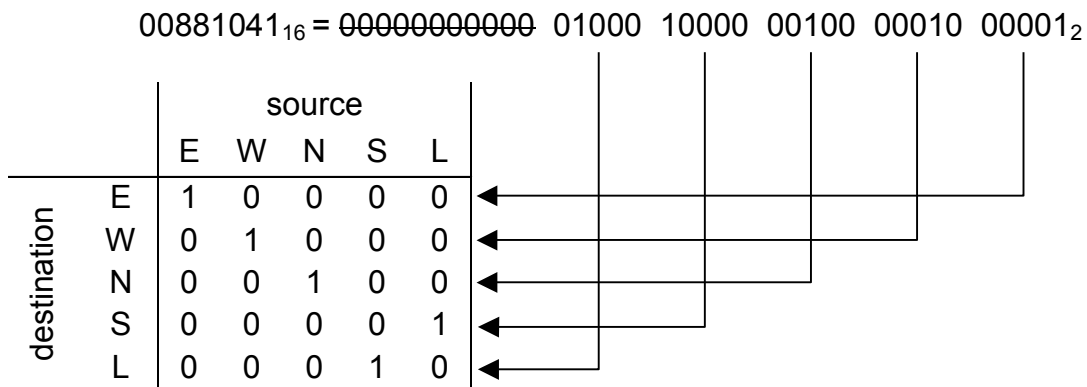


Fig. 5.20: Data from the block-RAM being transferred to a switching table.

5.6.2 Dual-layer NoC

The dual-layer NoC consists of two individual NoCs of type Hermes. The NoC allows the configuration of its main parameters in a single configuration file called HermesPackage. For this work the data-width and the buffer-size are of great interest. Whereas the data network is supposed to be fast with respect to high bandwidth and low latency, the performance of the control network is expected to be of no great interest. This is due to the assumption that the PR-process will be very slow compared to the NoC, meaning that the latency of the control network will not contribute much to the overall PR time. The control network is therefore optimized with respect to FPGA utilization.

The NoC needs to store the address of the packet's destination in a single flit. The address consists of the x- and the y-coordinate, for a three-by-three NoC this is two bits each, making it four bits in total. For this reason the minimum data-width for a flit is four. Fig. 5.21 shows the dependency of the FPGA utilization from the flit data-width. As expected the utilization, namely flip-flops and LUTs, increase with the increase of the data-width. For the control network the smallest possible data-width of four was chosen. In contrast to that the data-width of the data network directly contributes to the performance of the whole network. As a compromise between performance and precious FPGA resources, a data-width of eight has been chosen, being faster than a network with four bit, but saving 20% of FFs and 11% of LUTs compared to a data-width of 16 bits.

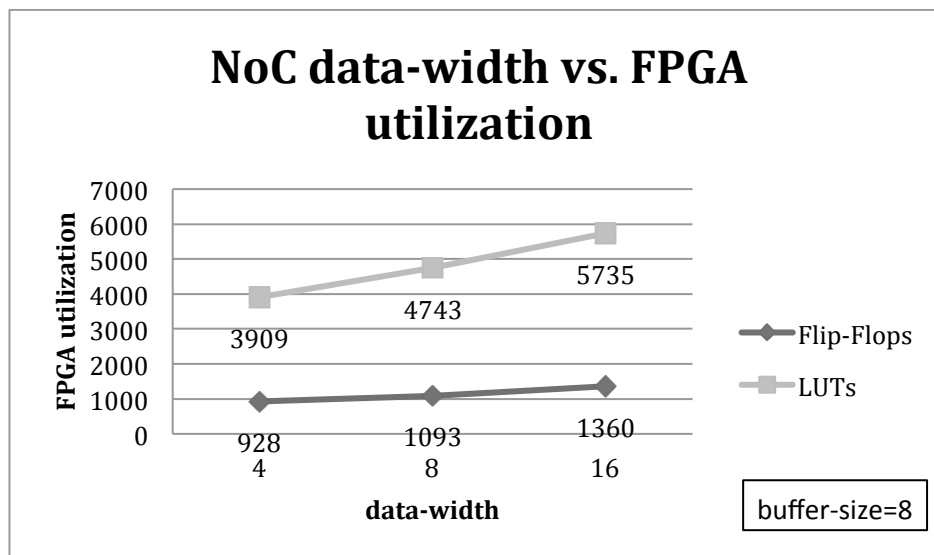


Fig. 5.21: Dependency of the FPGA utilization from the NoC data-width.

The other important parameter for a NoC is its buffer size. For the control network the buffer size can be reduced to its minimal size of one flit. As shown in Fig. 5.22 this saves about 3% of FFs and 24% of LUTs for the complete NoC. The reason for the savings of LUTs may be a result of the omission of the multiplexers for the buffers. A deeper analysis, for example with the FPGA-Editor, could clarify this. Nevertheless the choice for the smallest possible buffer size is obvious here.

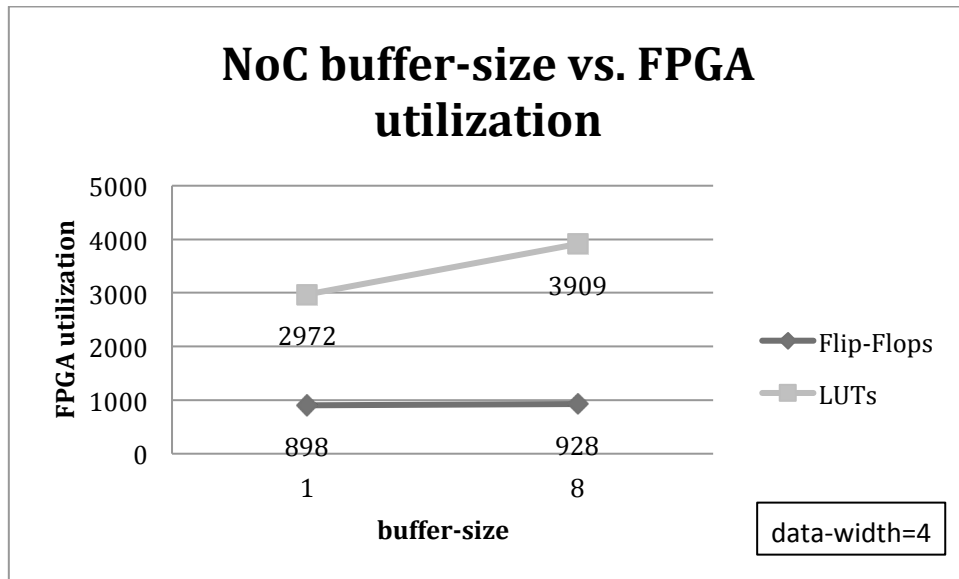


Fig. 5.22: Dependency of the FPGA utilization from the NoC buffer-size with a data-width of 4.

The buffer-size for the data network should be laid-out larger to allow maximum performance. With respect to latency the optimal buffer size for the Hermes NoC is six. Higher buffer sizes do not further decrease latency. Fig. 5.23 compares the FPGA utilization for different buffer-sizes. As expected a buffer-size of four results in less utilization than a size of six or eight. However, the use of eight flit buffers uses even less resources than the use of six flit buffers. This paradox result indicates that the real implementation seems to be more efficient for a size of eight. The internal representation with LUTs may profit from a buffer size which is a power of two. As a result of this, the buffer size for the data network has been chosen to eight.

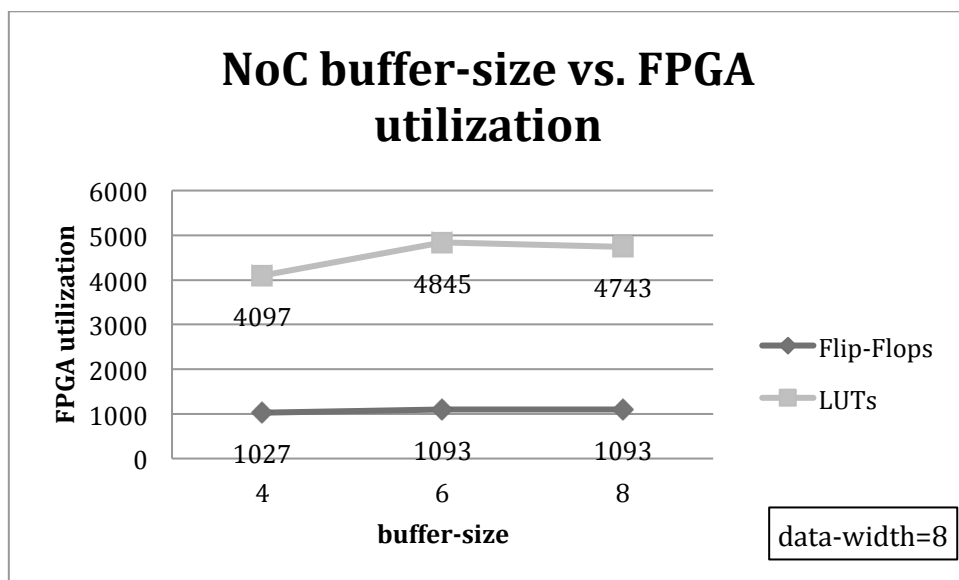


Fig. 5.23: Dependency of the FPGA utilization from the NoC buffer-size with a data-width of 8.

To allow the use of further versions of the NoC, beside the global parameters the NoC itself has been left nearly unchanged. However, as a result of high device utilization, the buffers have been further optimized. As they are instantiated three to five times per

router, for nine routers per NoC, even small enhances lead to huge savings. The one-flit buffer of the control network has been partly rewritten, substituting the RAM by a storage of a single flit and removing the pointer into the RAM. The pointer of the buffer for the data network could also be optimized. As the buffer-size has been chosen to eight, the pointer is three bit wide. For an arbitrary buffer size, the pointer must be reset to zero after reaching the end of the buffer space, resulting in a circular buffer. For a buffer size with a power of two, the pointer wraps around automatically back to zero. In this example from "111" to "000", with the carryover vanishing. This makes the reset unnecessary, again saving quite some resources.

5.6.2.1 NoC-Wrapper

The NoC-Wrapper builds the interface between the IP cores and the dual-layer NoC. Before an IP core can send any data to a different IP core, it has to request a route to the destination. To do so, it informs the wrapper with dedicated signals that it would like to establish a connection and specifies the destination. The wrapper then handles the request by forming and sending a packet to the PR_MASTER using the control network. When the PR-Master grants the route request, data can flow. The IP core itself can continue to send data using an already established route without interference. Once it tries to send a packet to a destination it has requested a route to, but is not yet established, the wrapper will detect this and stall the traffic until the route is established. This results in reduced complexity of the IP cores, as they do not have to wait for confirmation of the route. Fig. 5.24 shows the simplified FSM of the wrapper.

The FSM consists of two independent parts; the left part is responsible for receiving the route request from the IP core, forming a request packet to be sent to the PR-Master, and receiving its reply. The right part monitors the data flow of the IP core, stalling the traffic if needed. The send and receive states again work similar to the one of the Serial Interface Core described in Section 5.6.1.1, and the signals used for this communication will not be repeated here.

The left part of the FSM starts at the *rec* state after an initial asynchronous reset. It waits for the IP core to assert the *route_reqIP* signal high, indicating that it would like to establish or cancel a route. The *routeIP* signal then determines the destination to which it would like to establish a route, or the source from which it would like to cancel a route. The *estNcanIP* signal is high for the establishment and low for the cancelation of a route. All this information is stored within the FSM. If a route is to be established, the FSM sets the *route_requested* signal high and stores the destination IP core in the *route_requested_IP* signal. These two signals are later used to decide whether a data packet is sent using a not yet established route, which must be stalled.

The FSM then forms a packet of six flits to the PR-Master with the information just received from the IP core. The structure of this packet is described in Fig. 5.24 at the lower left. The FSM then goes through the states *send_1* to *send_6* with the according acknowledgment states *send_1_ack* to *send_6_ack*, sending the packet on the control network. When a route had to be canceled, no further steps are needed and the FSM

returns to the initial *rec* state. When a route had to be established, the FSM has to wait for the reply of the PR-Master, either acknowledging the establishment or denying the request. For this the FSM receives the four flit answer packets in the states *rec_header_pr_reply*, *rec_size_pr_reply*, *rec_routing_possible_pr_reply* and *rec_data_pr_reply*. The third flit is of greatest interest: if it is one, the routing is possible, and the FSM sets the *stall_traf_reset* signal for one clock cycle. Then it returns to the initial *rec* state. If it is zero, it means that the route could not be established by the PR-Master. Then the FSM goes into the *wait_for_rerequest* state, remains there for a certain time, and then goes back to the *send_1* state, requesting the route again.

The right part of the FSM basically monitors the traffic from the IP core to the data network. It starts at the *rec_dst* state after an initial asynchronous reset. At reset, the *glitch_remove* signal is asserted high. Its function will be described later. The FSM then waits for the IP core to assert the *txIP* signal high. When this happens, it moves to the *rec_dst_glitch* state, asserting the *glitch_remove* signal low. Then it waits for the data network to accept the flit by asserting the *ack_rxCN* signal high. The FSM then goes to the *rec_size* state, where it receives the size flit when the IP core send it and the data network acknowledges it. It also sets its internal size counter to one. Then the FSM loops in the *rec_data* state, receiving all data flits. It determines the end of the packet by counting the flits and comparing the counter with the previously received packet size. When all flits are received, it returns to the initial *rec_dst* state, again asserting the *glitch_remove* signal high.

Now the FSM can determine when a packet has to be stalled as it is sent to an IP core to which a route is not yet established. For this it evaluates the following signals by logic. First, the *route_requested* signal must be high. If no route is currently requested, there is no need to stall any data. Then the *state_watch* signal is evaluated. Only the first flit of a packet needs to be stalled, as it contains the address of the destination IP core. Therefore stalling is only needed in the first two states, namely *rec_dst* and *rec_dst_glitch*. The first flit is then compared to the *route_requested_IP* signal, which stores the address of the IP core to which a route is going to be established. If they do not match, again no data has to be stalled. Finally, the *stall_traf_reset* signal from the left part of the FSM is evaluated. Once the PR-Master confirms the establishment of the route, this signal is asserted high, and the data may flow again.

The FSM therefore stalls the very first flit by one clock cycle if a route is requested but not yet granted. Unfortunately, the first implementation of the FSM produced a glitch of the *txIP* signal. As the *txIP* signal is forwarded using combinatorial logic only, it is possible that it is forwarded just before the blocking part of the logic becomes active. This results in a very small glitch, which might be enough to trigger the following stage. To circumvent this, the *glitch_remove* signal was introduced that masks that glitch which occurs at the time the right FSM enters the *rec_dst* state.

5.7 Early Access Partial Reconfiguration design flow

Before entering the EAPR design flow, a regular non-pr version of the design should be

implemented and tested on the FPGA. This avoids resolving simple design issues within the time-consuming PR-implementation. Once the non-pr implementation works as expected, the pr-version can be implemented. If it does not work as expected, the error is limited to the PR parts and can be resolved more efficiently.

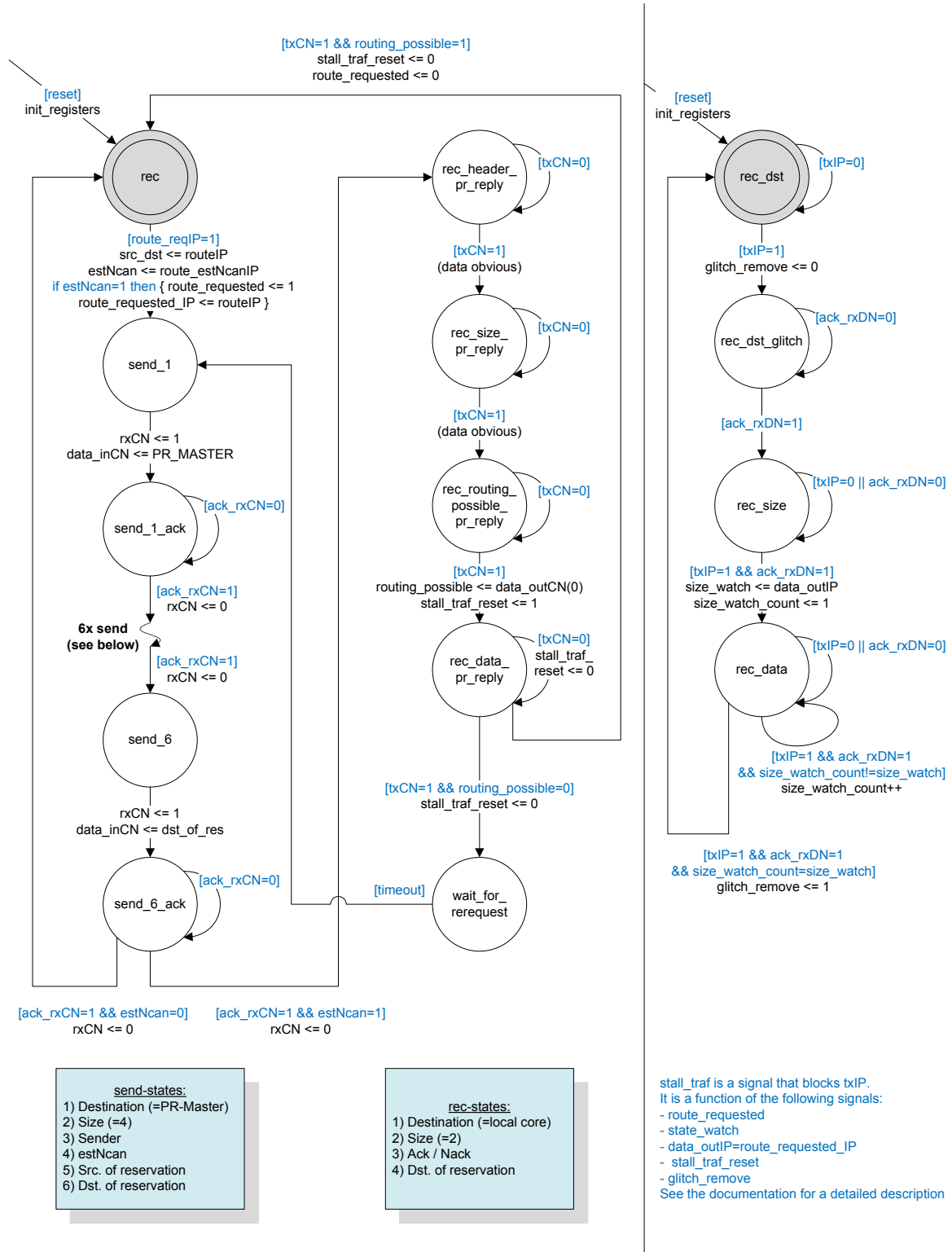


Fig. 5.24: Simplified finite state machine of the wrapper module.

To implement a non-pr version, one have to create a new ISE project and chose one PRM per PRR. This PRM then cannot be changed after implementation, so it should be chosen wisely. For the dual-layer NoC, a router has been chosen which satisfies all route request for an exemplary packet-path. Also the PR-Master has to be deactivated so it does not try to PR the FPGA at runtime. For the dual-layer NoC it is sufficient to modify the PC-program responsible for the PR using the JTAG interface. In addition to this, a jumper on the board was set up that lets the PR-Master skip the serial request for PR and directly jumps to the confirmation state.

Once the non-pr implementation works as expected, the PR version can be implemented as being described in the following Sections.

5.7.1 Synthesis

Before starting the synthesis process, one should make sure that the right system type is selected. The package SoCGeneral contains a globally available constant `SYSTEM_VERSION` that can have the values `REGULAR`, `PR` or `PR_BUFFER`. This constant controls which of the three available versions will be implemented. The impact of this constant is shown in Fig. 5.7 and described in detail in Section 5.5.

Now the parts of the PR design can be synthesized independently. For this it is advisable to create a separate folder, here named *synth*. There an individual ISE project is created for every part, namely the top module, the individual PRMs and the static module (here a second static module exists for the buffers of the PR-router and a third one for the push-button debouncer). The projects all reference the source-files from the source directory. The top module will only black-box instantiate the underlying modules and merge with them at a later stage. For all but the top module the projects also have to have the I/O-buffers disabled. Otherwise ISE will add I/O-buffers and connections to arbitrary output pins. The option can be found by right clicking on the "Synthesize - XST" property, selecting "Properties" and choosing "Xilinx Specific Options" in the appeared window. In addition to this, for all project the "Keep Hierarchy" option must be enabled, which can be found in the same window under "Synthesis Options".

If more than just a few PRMs exists, it becomes impractical to perform the synthesis by hand for every module. For this, the program `SoC_Ramgenerator` described in Section 5.5.3.1 does not only generate the 120 PR-routers, but also a batch-file for synthesis. This batch-file is designed to copy the router source-files individually to the synthesis directory, execute the ISE synthesis tool XST, and then move the resulting synthesized file to a directory with the routers name. This is repeated for every of the 120 routers. The synthesis uses the options previously selected in the ISE project itself.

5.7.2 Floorplanning

Floorplanning defines the physical positions of the individual modules or parts on the FPGA. It can be performed on a text-bases using a UCF file, but it is much easier to use

PlanAhead which visualizes the process. PlanAhead can then also be used to automate the PR generation process and is therefore the tool of choice. Unfortunately, PlanAhead in version 10.1 becomes very unstable once a project is set as being partially reconfigurable. It then crashes quite often, predictable and unpredictable on certain tasks. Therefore it is best to use two separate PlanAhead projects. The first one is used for floorplanning only, where the project is not set to be partially reconfigurable, whereas the second one is used for the PR automation only and inherits the floorplan from the first project. This reduces the number of program-crashes to an acceptable level.

For the first project one can use the non-pr version of the design and create a PlanAhead project from it. At first the existing constraint-file with the pin-locations and timing constrains can be imported. Then the actual floorplanning can be done by creating new Pblocks (physical blocks). Every Pblock can have one or more modules from the netlist assigned to it. In the case of the SoC with a three-by-three array of IP cores and according routers and wrappers nine Pblocks have been created, arranged in a three-by-three matrix fitting the SoC. The Pblocks can consist of basically all elements of an FPGA like CLBs, RAMs, DSPs and DCMs. Then the modules from the netlist are added to the Pblocks. Here every Pblock gets an IP core, a wrapper, a control router and a data router assigned to. This keeps the individual parts in the matrix separated, but allows the local parts, namely the IP core, its own wrapper and the directly attached two routers to merge. The merging saves some resources on the FPGA, while the global structure stays intact and neighbor routers stays next to each other, keeping wires short and possible frequencies high. In addition to the Pblocks, the BUFGs, the DCMs and the BMs should be fixed in place.

Fig. 5.25 shows the placement of the complete SoC on the FPGA. Due to the large power-pc block at the bottom left of the FPGA (black rectangle) the CLB structure becomes a bit irregular. In addition to this, the IP cores as well as the routers have different sizes, resulting in a slightly irregular structure of placement. The individual Pblocks, labeled as *area_00* to *area_22*, are basically still placed in a three-by-three matrix. *Area_20* is highlighted, showing the four containing modules also highlighted at the netlist on the list side.

The placement of the central router, which is the PR router, is of special interest. As it was described in detail in Section 5.1.2.2 the size and placement of the PRR should be chosen wisely. This minimizes the size of the partial bitstream files and speeds up the PR process. The PRR is the high but narrow region in about the middle of the FPGA, aligned along the clock-regions. The BMs, which straddle the border of the static- to the PR-region, can be seen as red dots. Fig. 5.26 shows the PRR magnified with one BM highlighted.

The placement can finally be exported into an UCF file, holding the area constraints in addition to the pin and timing constraints of the very first UCF file. This new constraint file can then be used for the implementation of the non-pr version including placement constrains as well as for the PR-design in a separate PlanAhead project, as will be

illustrated in the next Section.

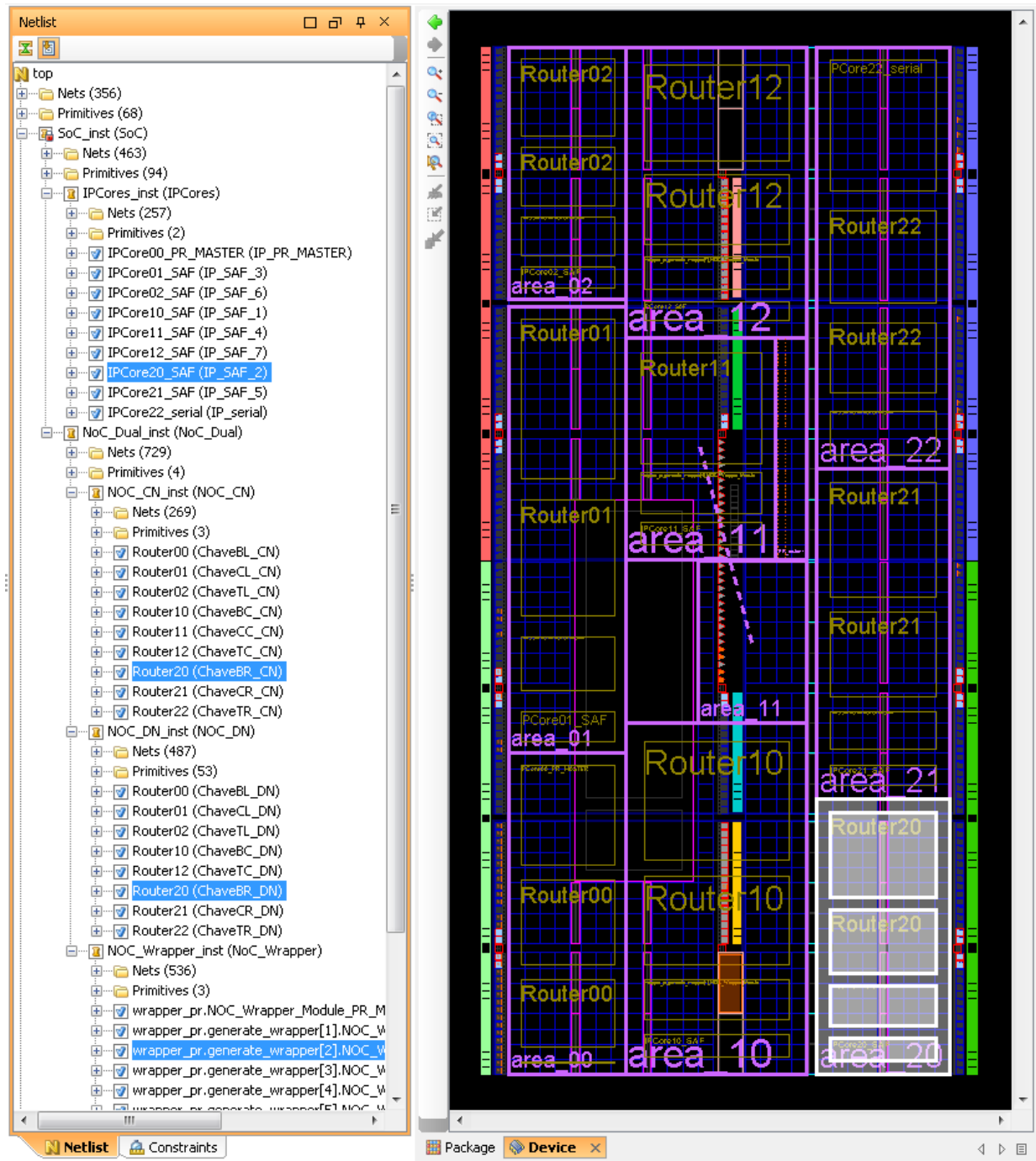


Fig. 5.25: Floorplanning with PlanAhead - placement of the modules in Pblocks.

5.7.3 PR Implementation

The synthesized files together with the basic floorplan can finally be implemented to form the PR project by PlanAhead. For this a new project is created and the synthesized netlists are imported. The top module is the main file, and the directories containing the static netlists and the BMs are added to it. One arbitrary PRM per PRR must be added as well, the other PRMs will be added later. The import dialog then look like Fig. 5.27.

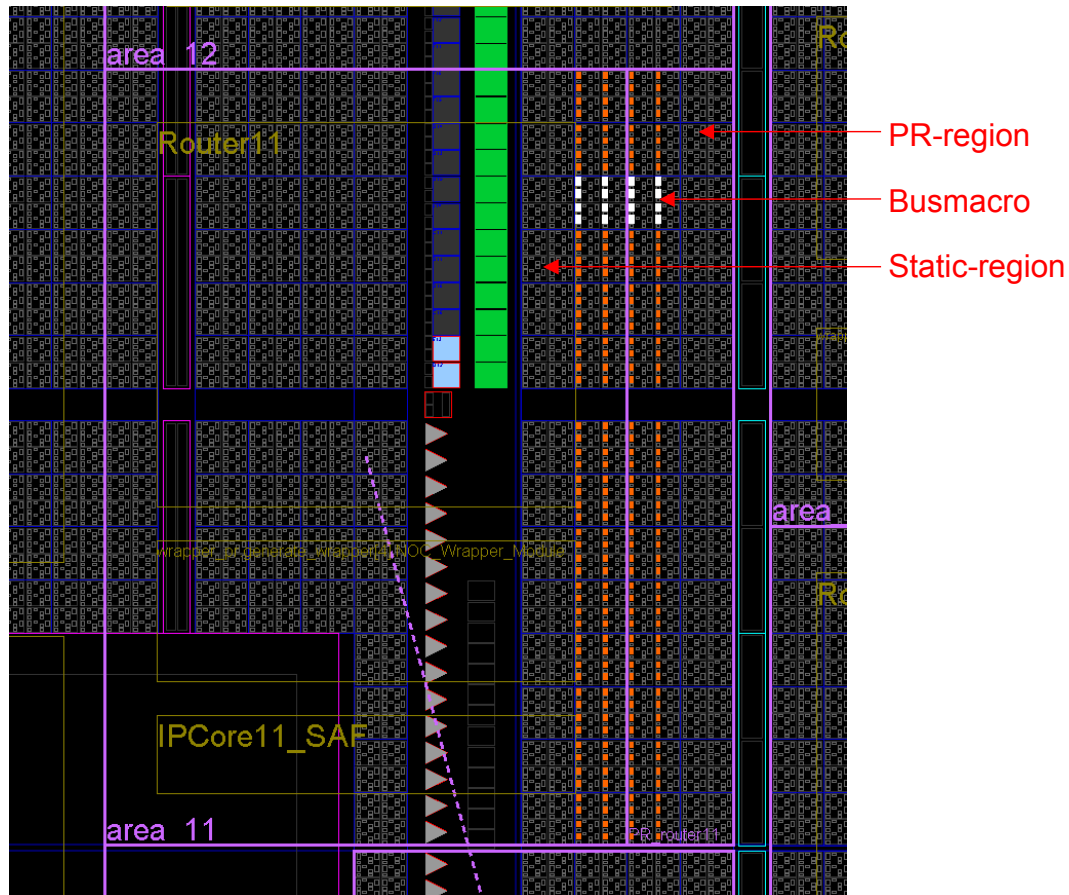


Fig. 5.26: Floorplanning with PlanAhead - PRR magnified with one BM highlighted.

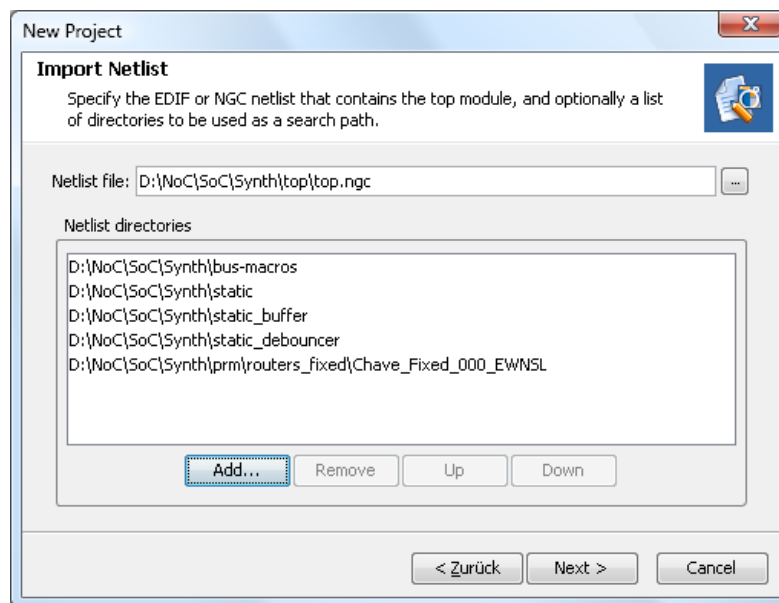


Fig. 5.27: Importing netlists to PlanAhead.

Afterwards the correct FPGA is chosen and the constrain file generated in Section 5.7.2 is added, which contains the pin placement and the main area constraints. Then PlanAhead opens and one can choose "Set PR Project" from the File menu. Afterwards the `pr_router` in the netlist window can be set as reconfigurable. The module name should be chosen to `Chave_Fixed_000_EWNSL`. PlanAhead generates a new Pblock

with the name `pblock_pr_router.Router11` as the PRR. Then the old Pblock for the central router can be deleted and replaced by the new Pblock by right-clicking on the PRR and selecting "Set Pblock Size".

Now the other PRMs can be added. Again, this process can be automated by the `SoC_RamGenerator`. For this it has the button "Generate PlanAhead-data", which generates a script that adds the 120 routers (the first one should be omitted, as it is already part of the project). The script can be copied to clipboard and edited with a text-editor if needed. Then it is pasted in the console window of PlanAhead. After a while, all 120 PRMs are available in PlanAhead. Then the initial module can be selected, which will be later part of the static bitstream file. This is done by right-clicking on it in the netlist windows and selecting "Set as Active Reconfigurable Module".

Finally the implementation can take place. For this at first the static design is implemented by right-clicking on the word "static" in the "ExploreAhead Runs" window and selecting "Launch Runs...". This implements the complete static design, leaving the area marked as PRR empty. Then the PRMs are implemented by selecting them and also starting the "Launch Runs..." dialog. This will implement the PRMs within the PRR, ignoring the static design. For both processes the BMs work as a fixed endpoint. They guarantee that the static- and the PR-module stay pin-compatible. Finally the static and the partial parts are merged by right-clicking into the "ExploreAhead Runs"-window and selecting "Run PR Assemble...". This will generate a full bitstream file with the previously selected PRM embedded and a partial bitstream file for every PRM. The full bitstream file can then be loaded into the FPGA by using impact. PlanAhead should then finally look like Fig. 5.28.

The complete implementation by PlanAhead takes about four hours on a modern PC. This makes quick tests with a non-pr version as described in Chapter 5.7 so important.

5.8 SoC simulation and test

5.8.1 SoC simulation

A PR project is difficult to simulate and cannot include the PR process itself. However, most of the SoC can be tested in simulation when a packet is chosen for which its path can be completely covered by a single PRM. Then this PRM can be used for the complete simulation and the PR process is simply omitted.

The testbench for the SoC is depicted in Fig. 5.29. It consists of two completely independent FSMs, one responsible for sending and receiving packets using the first serial interface, the other responsible for receiving and confirming PR requests using the second serial interface. The clock and reset generation of the testbench is omitted for simplicity. The testbench cannot perform the PR process as it is not capable of modifying the SoC. The figure therefore shows the JTAG-interface of the corresponding FPGA implementation as not connected, resulting in the restriction of using a single chosen PRM.

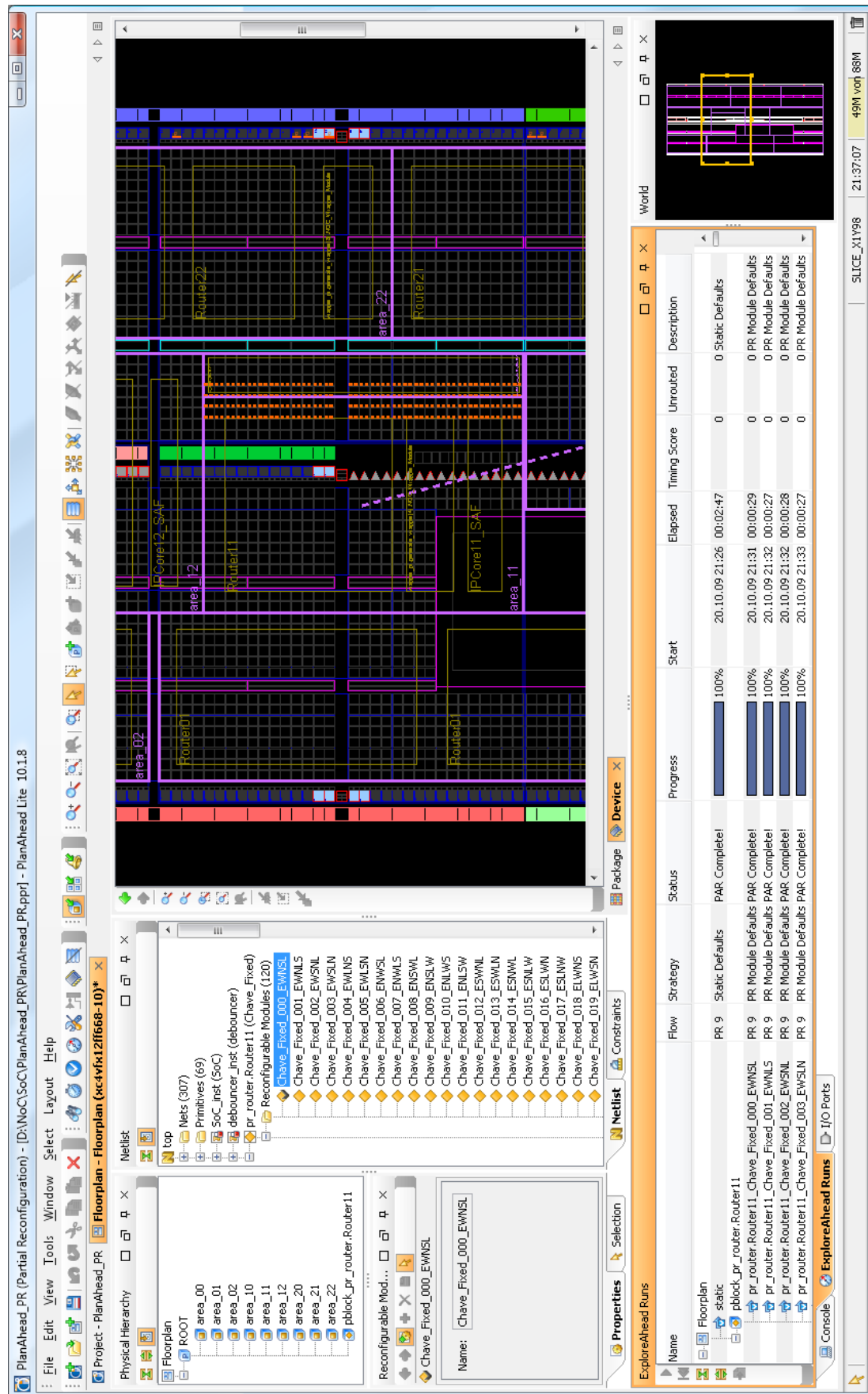


Fig. 5.28: PlanAhead after successful implementation of the PR design.

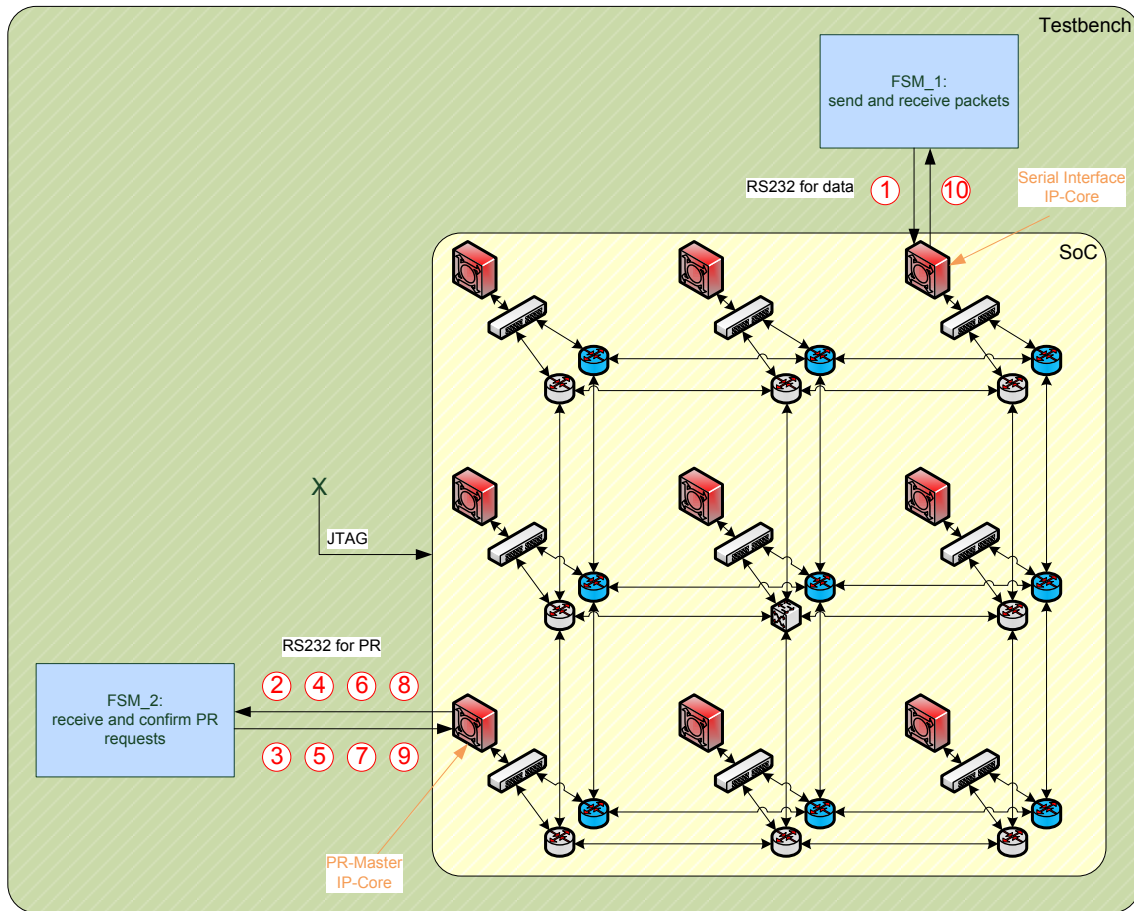


Fig. 5.29: Overview of the testbench with the main signals to the instantiated SoC.

The first FSM is used to insert data into the SoC. For this it uses an array of flits that represents one or more packets. In addition to this the very first flit of this array is $55_{16} = 0101\ 0101_2$, which is a synchronization byte that is used by the autobaud part of the serial interface IP core. Using this byte, which contains alternating zeros and ones, it automatically adjusts to the baudrate of the sender. The FSM then sends the individual flits, which are also one byte each. The FSM simulates the behavior of a real RS232 serial interface so that the complete SoC including its serial cores can be tested. For each byte of data the FSM sends a start-bit, the eight data-bits and a stop-bit; no parity bits are used here. Then the FSM loops through all data flits. It is important to notice that the simple version of the RS232 interface is used. This uses two wires only, one for sending and one for receiving, and does not provide any kind of flow control. This works well here as the serial interface is much slower than the SoC, and therefore forms the bottleneck anyway. The FSM also receives packets sent from the SoC, but does not decode or react on them.

The second FSM is connected to the PR-Master. It also starts by sending the 55_{16} synchronization byte. It then waits for PR request on the serial link. As it cannot perform any PR process, it simply acknowledges every requests by returning an FF_{16} . The format and timing for the serial data is identical to that of the first FSM.

Fig. 5.29 also shows the typical order of events for an example packet. At first (1), the

packet is sent to the SoC by the first FSM. Then the packet travels along the network in its pre-defined path, which results in some route establishment and cancelation requests. Every time the PR-Master decides a new PRM is needed, it requests it by sending the number of the PRM to the second FSM (2)(4)(6)(8). This then replies on every request (3)(5)(7)(9). Finally, the packet arrives at the Serial Interface IP core and is sent back to the first FSM (10).

5.8.1.1 Example packet

The example packet used in this and the following Sections is "A2,04,04,05,01,0A". Its destination is IP core with #2 (right half of first flit) and sent by IP core #A. The second flit is its size of four data flits (excluding header). The following flits then describe the path the packet will take, namely IP core #4, #5, #1 and back to #A. The packet therefore goes from the Serial Interface IP core to four Shift-And-Forward IP cores before returning to the Serial Interface IP core. There it enters and leaves the SoC.

The addresses of the cores are a result of the naming conventions described in Section 5.6.2. For example a core with the coordinates $x=2_{10}=10_2$ and $y=1_{10}=01_2$ will have the address $xx:yy=1001_2=9_{16}$. The addresses of all cores are shown in Fig. 5.30. It also shows the path the example packet takes through the network.

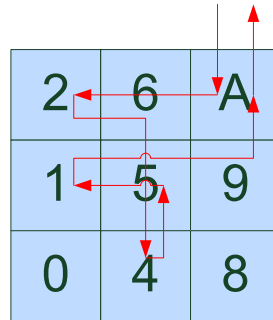


Fig. 5.30: Addresses of the IP cores and path of the example packet.

The packet expected to leave the SoC should have the destination of #A and the sender of #1, as these are the last two cores it passes. Its size should be still four. The rest of the packet is also straight forward when taking into account that it was modified by every of the four Shift-And-Forward IP cores it passed. The modification method, basically a left-shift, was described in Section 5.6.1.2. The packet to be received is then "1A,04,02,04,05,01".

5.8.1.2 Simulation with Modelsim

The simulation of the SoC with the example packet described in the last Section was performed using Modelsim. For the simulation to work correctly, the time resolution must be set to 1ps. Otherwise, the DCM within the SoC will fail with random and bizarre errors. The signals for the serial interfaces then looks like Fig. 5.31. On the very left the two synchronization bits can be seen, followed by the example packet on the top. Afterwards the four PR request of the bottom signal can be seen as #2, #3, #9 and #24

(the numbers can be read out from the signal, even though this is not clearly visible in the figure). Each PR request triggers a response in form of an acknowledgement code, seen in the second-bottom signal. As this is FF_{16} , only the start-bits are visible. Finally the packet, modified on its way, is sent back.

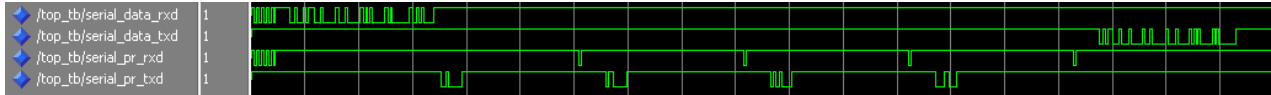


Fig. 5.31: Simulation of the serial interfaces.

The simulation shows exactly the expected behavior of the SoC. The packet it returns on the serial data connection is as expected. The PRM that are requested are also valid. For example the first request is PRM #2, which is a router connecting the north input to the south output, as needed. Details about the routers have been described in Section 5.5.3.

5.8.1.3 Buildup and test of the system

The test system is built up from the ML403 evaluation board as described in Section 5.5.1. As this board offers only one serial interface, a second one is attached in form of a simple level-shifter with a MAX3232 chip. This is then connected to the general-purpose I/Os of the FPGA. Due to the lack of RS232 interfaces of today's PCs, the two serial interfaces are then connected to the PC using RS232-to-USB converters. There they can be accessed as regular COM-ports.

For the data link to the FPGA a Java program called SerialApp is used. It connects to a COM-port and allows to transmit and receive data using it. Fig. 5.32 shows the program with the example packet and the response of the SoC.

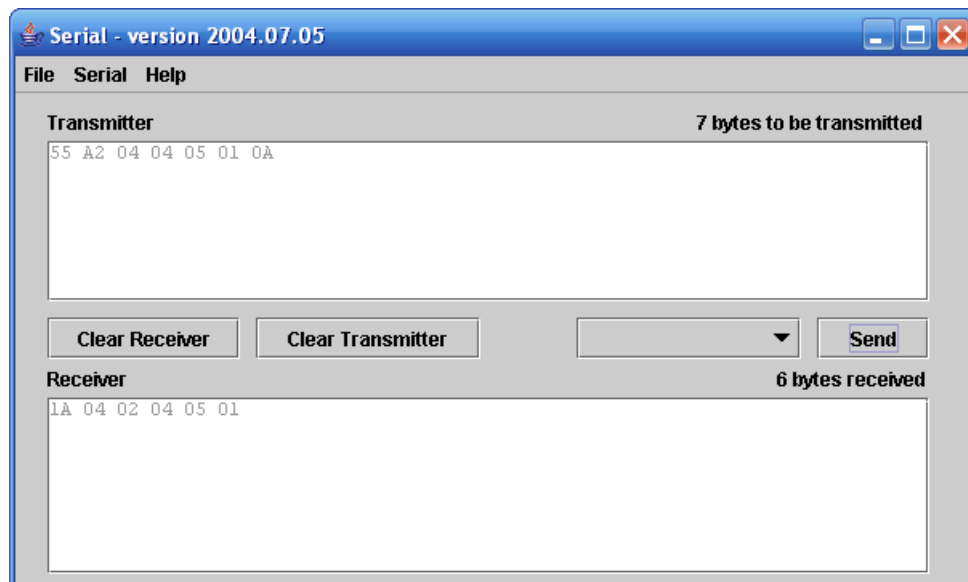


Fig. 5.32: The program SerialApp, used to transmit and receive data using a COM-port.

The serial link for the PR-process is handled by the program "Serial Reconfiguration" written in C#. The program receives a PR request containing a number between 0 and 119 on the COM-port. It then invokes impact, part of ISE, to partially reconfigure the FPGA using the JTAG interface. After completion of the process, it acknowledges this

back to the SoC. The complete process takes about two seconds as impact needs some time to find the attached FPGA programmer. The PR process itself is much faster as will be shown in Section 5.9.2. Fig. 5.33 shows the program with the COM-port parameters on the left and the directory selections on the right. The program has initialized the serial port and sent out the synchronization byte. Then it received four PR requests and performed the PR process for each.

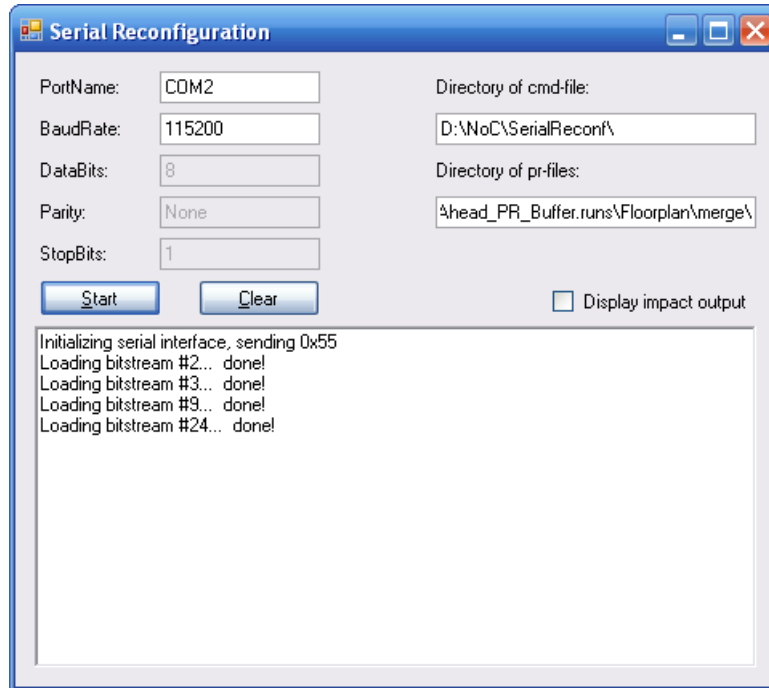


Fig. 5.33: The program Serial Reconfiguration, used to induce the PR-process.

The test shows that the SoC behaves as expected. Simulation and test matches perfectly. The packet received is as expected, and the PR requests also match the simulation. The PR process works successfully and lets the packet travel through the reconfigured region. The opposite is also valid. Wrong PRMs loaded into the FPGA prevents the packet from reaching its destination, showing that the PR process is important and successful.

5.9 Timing analysis of the SoC

5.9.1 Timing analysis in simulation

A more detailed view of the SoC can be revealed by the simulation of internal signals. This is most important at the design phase to compare the expected behavior of parts of the system with the behavior of the really built system. Differences indicating problems can there be tracked down to their sources and the issues then corrected. The simulations of the three different IP cores have already been shown in Section 5.6.1. Now the focus lies on the timing between the different components. For this the example packet from Section 5.8.1.1 is used and the following timings of the packet moving from the top-left IP core to the bottom-center IP core is analyzed:

1. cancelation of the old route (using the control network)
2. establishment of a new route (using the control network)
3. packet traveling from source to destination (using the data network)

For implementation issues the enumeration of the Hermes NoC routers (and therefore also wrappers) is a continuous sequence from zero to eight, starting at the bottom left and numbering horizontally first, vertically second. Unfortunately this does not match the addresses in form of the axis coordinates $xx:yy$ used so far. Fig. 5.34 shows the continuous enumeration used in this Section and the three timing steps that will be analyzed.

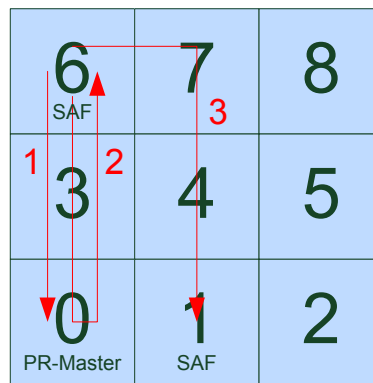


Fig. 5.34: Continuous enumeration of IP cores and timing steps to be analyzed.

Interesting is the latency the individual modules cause. To measure this, the time of each module on the way raising its tx signal is recorded. The latency of a module is then the difference in time of it raising its tx signal from the time of the previous module raising its tx signal.

Fig. 5.35 shows the timing analysis for a packet traveling from IP core 6 to IP core 1. The SoC shown is the PR version without dedicated buffers. The three individual steps as mentioned above are separated by markers. The clock frequency of the simulation has been chosen to 10Mhz, which makes a clock cycle $0.1\mu s$ long. The total time for the packet, from the sender asserting first its tx signal high until the receiver acknowledging the last flit can be seen in the figure. It is the sum of the time of the three steps, $12.3\mu s$ in total, which is equivalent to 123 clock cycles.

Nevertheless, the timing of the simulation may not be identical to the timing of the real system. The use of a handshake mechanism allows signals with long propagation delays to arrive at a later clock cycle without making the system fail. This can hardly be shown in simulation. Therefore it is more accurate to analyze the real system instead.

5.9.2 Timing analysis of the real system

For a deep test of the system and a better understanding of internal processes a closer look into the running system is desirable. An interesting approach to measure real signals inside an FPGA is the use of the ChipScope Pro Analyzer software from Xilinx. It

consists of a core module that is added to the ISE project and can connect to any internal signal. It is then synthesized and implemented together with the design. When the resulting bitstream is loaded into the FPGA, the core can be accessed via the JTAG interface and data can be acquired. The data is first stored in a Block-RAM in real-time, and then transmitted to the PC where it can be viewed. This is a very nice solution to measure internal signals as it does not require any additional hardware. Unfortunately, the ChipScope Pro Analyzer fails when the PR software overlay is installed. Therefore it cannot be used in the current version for PR designs.

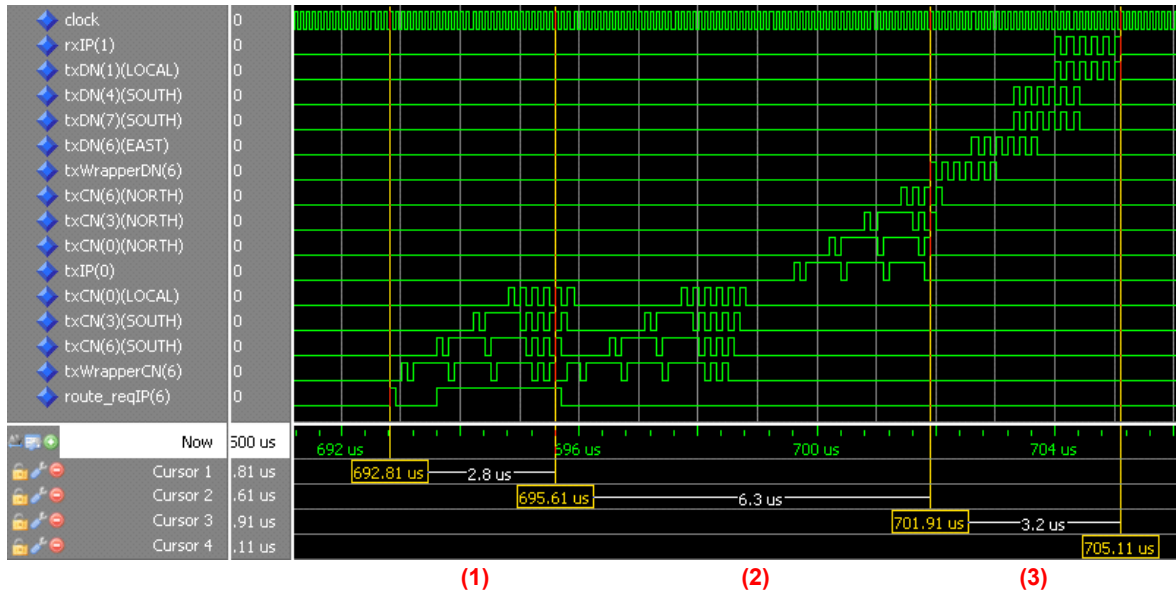


Fig. 5.35: Timing analysis showing the time specific modules raising their tx-signals.

An alternative approach is the use of a logic analyzer. This external device can connect to any available pins and capture data from them. For this it is necessary to route the signals to be monitored to the top-level module, and let them exit on the general purpose I/Os. The ML403 evaluation board connects many of them to an expansion header, which can be then connected to the logic analyzer. The logic analyzer works similar to a digital storage oscilloscope. It can trigger on a specific signal event and then records data from its inputs for a certain time. The data can then be viewed and analyzed as needed. The core difference to a digital storage oscilloscope is that the logic analyzer only samples binary values.

5.9.2.1 Timing analysis of the signal flow

For the timing analysis of the SoC the same *tx* signals are used as for the simulation shown in Section 5.9.1. This allows direct comparison of the simulation with the real system. It also allows to find problems that does not occur in the simulation. This includes problems regarding propagation delay and timing violations, but may also be a result of wrong assumptions of the simulator. The logic analyzer can also show details of the handshake signals, which are not visible in simulation.

Fig. 5.36 shows the measurement of the *tx* signals of the real system for the PR version without dedicated buffers. Their timing is identical to the timing of the simulation. This

shows that the real system works as expected. The comparatively low frequency of 10MHz together with a global clock for all modules obviously lets the handshake signals work in a synchronous manner.

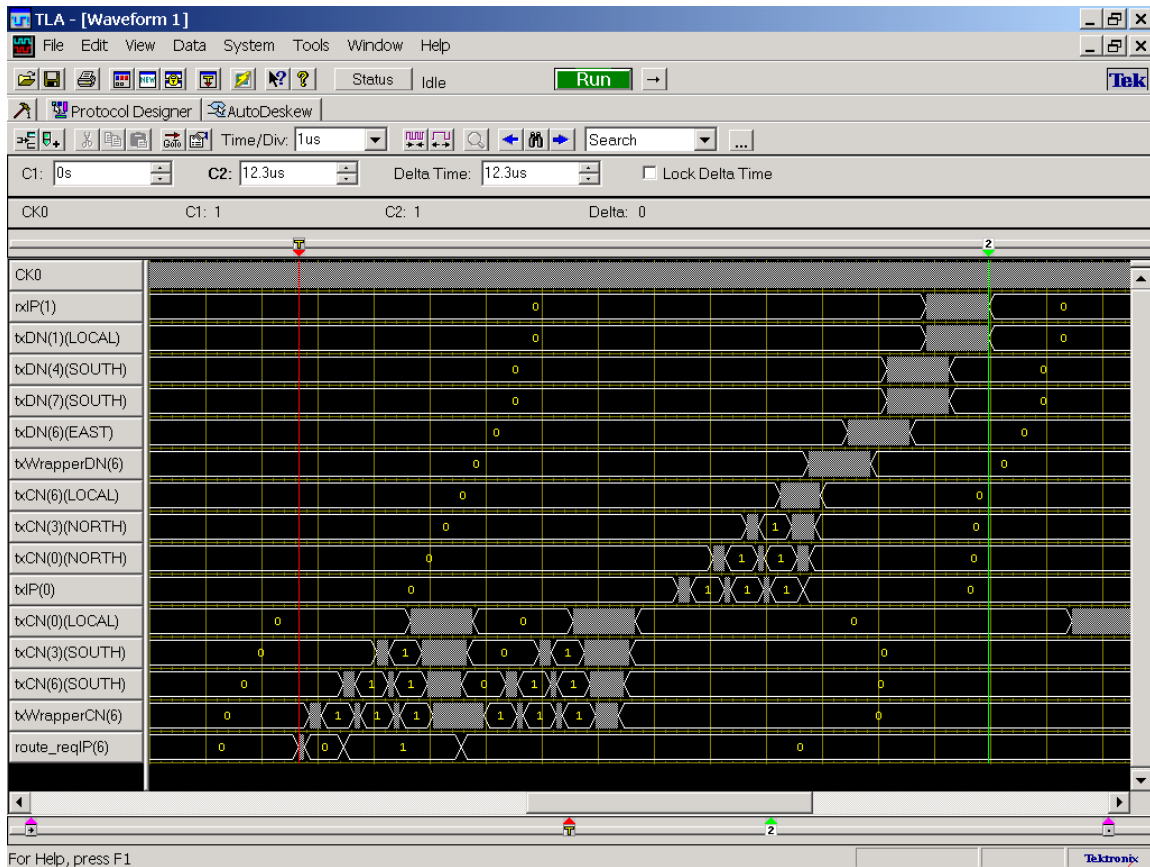


Fig. 5.36: Measurement of the real system – complete.

Now the latency of each module can be determined. The first part, the cancelation of the old route, is shown in detail in Fig. 5.37. At time zero the Shift-And-Forward IP core 6 raises its *route_req* signal (red marker). Two clock cycles later, the wrapper reacts and raises its *tx* signal (blue marker). One clock cycle is needed for the recipient of the request, and one for starting the sending. Then the packet travels through three control routers until reaching the PR-Master. Each control router takes six clock cycles for routing and switching (green and two purple markers). As the PR-Master uses the cancelation packet only for its internal table and does not reply to this packet, this path ends here.

The second part is the establishment of a new route to the new destination of the packet. The establishment packet is of the same type as the cancelation packet and would normally cause exactly the same latency in each module. As the IP core tries to perform this directly after the cancelation, the wrapper has to finish sending the cancelation packet first before it can form an establishment packet. This can be seen in Fig. 5.37 as the second *route_req* signal from IP core 6 is not acknowledged for quite some time. Afterwards the timing for this packet until reaching the PR-Master is the same (not shown).

Then the PR-Master has to find an appropriate PRM and load it into the FPGA. The searching process can take different times, as the PR-Master goes through the 120 possible PRMs and stops when it finds an appropriate one. This may be just the next, or it might be the 119th. The PR-Master needs one clock cycle for each test. It is also possible that the currently loaded PRM fits and no new PRM needs to be loaded. Nevertheless the searching takes much less time than the actual loading of the PRM, and therefore can be basically neglected. In this example the loading step is simply skipped by having a jumper installed at the non-pr pins. A PRM valid for all route requests have been previously chosen. In total the PR-Master has a latency of 19 clock cycles here, but this value is not representative and does not include the loading of the PRM.

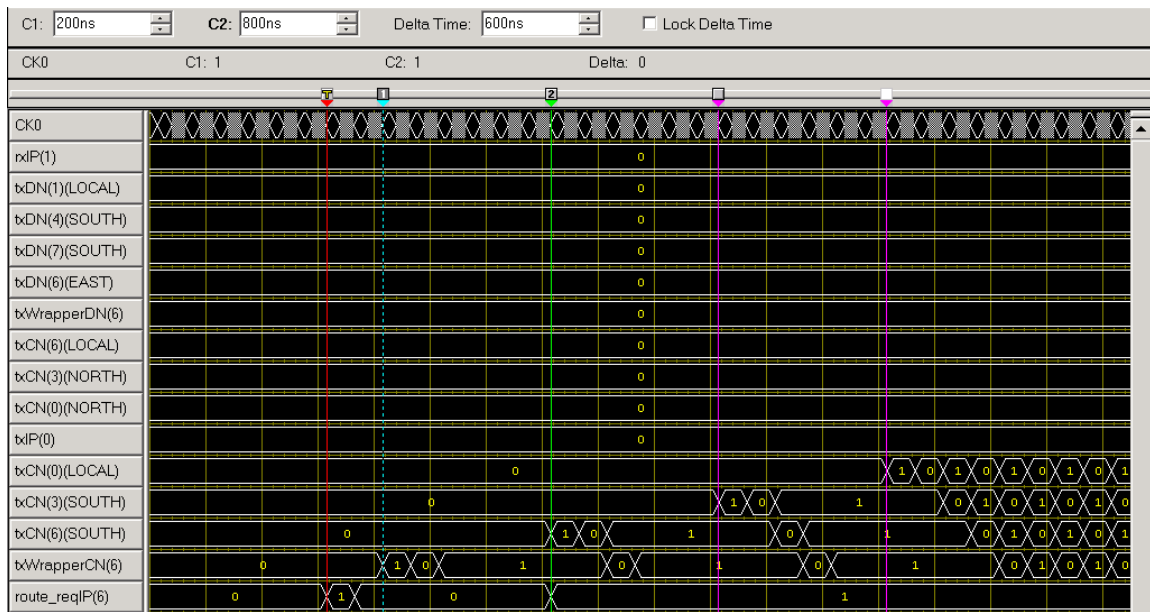


Fig. 5.37: Measurement of the real system - route cancelation.

In the next step the acknowledgement packet travels from the PR-Master back to the wrapper. This is shown in Fig. 5.38. It starts with the PR-Master raising its *tx* signal (blue marker), and continues with the forwarding by the control routers, again taking six clock cycles each (green and two purple markers).

Then the wrapper receives the acknowledgement packet, as shown in Fig. 5.39. The third flit indicates that the route establishment has been successful, and no retry is needed. Just after receiving this flit (red arrow), the wrapper stops stalling the data packet from the IP core and allows it to enter the data network (green marker). This takes five clock cycles. The fourth flit, representing the destination of the reservation, is of no use here and could be optimized away. At the beginning it was designed to let the wrapper request multiple routes at a time, but this has not been implemented due to its higher complexity.

Finally the data packet can take its way from the source to the destination, using the data network. This is shown in Fig. 5.40. Each marker shows one router starting to send. Interestingly, this takes seven clock cycles each, one more than for the control network. This is a result of the buffer optimization described in Section 5.6.2. The buffers of the

control network can store one flit only, and therefore need no RAM with pointers into it. Incrementing these pointers and reading out the data from the RAM takes an additional clock cycle compared to a dedicated register with no pointer. Important to notice is that the PR-router with number four does not have any (visible) latency. This is due to the fact that it has hard-coded paths directly from inputs to outputs, and no routing or switching takes place. This version also does not use buffers for the PR-router.

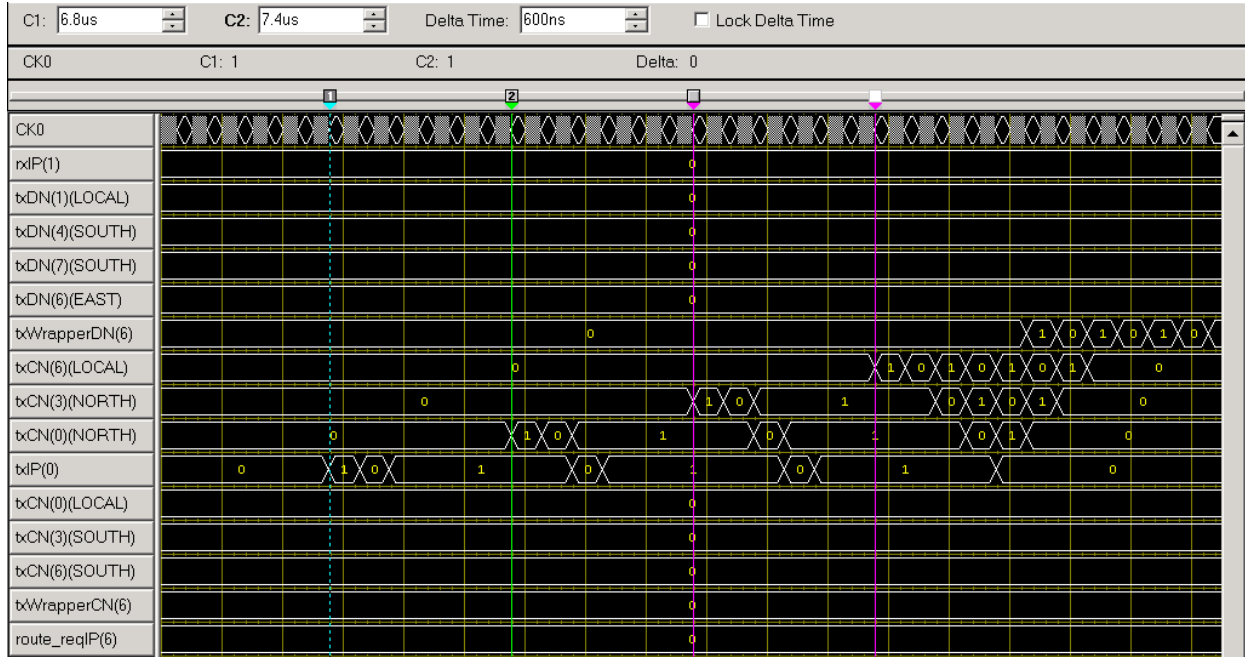


Fig. 5.38: Measurement of the real system - route establishment, acknowledgement packet.

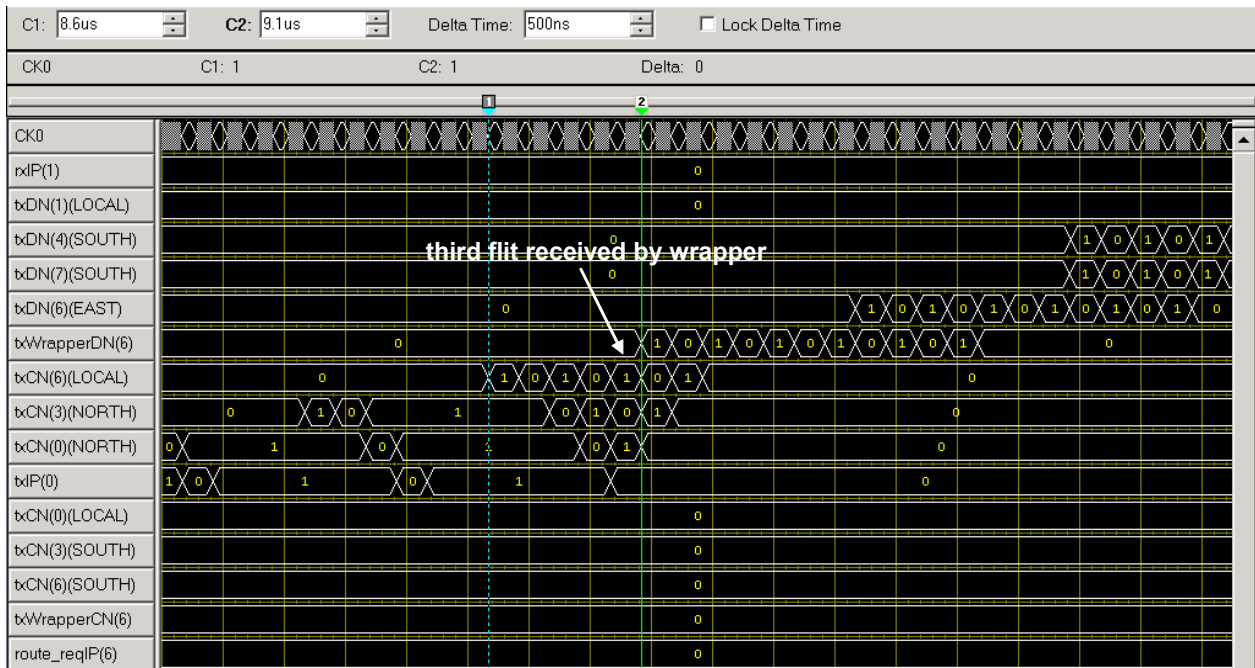


Fig. 5.39: Measurement of the real system - wrapper stops stalling the data packet.



Fig. 5.40: Measurement of the real system - data packet traveling through the data network.

Finally the data packet arrives at the destination IP core (blue marker) and is completely received there (green marker). This takes 11 clock cycles, as can be seen in Fig. 5.41.

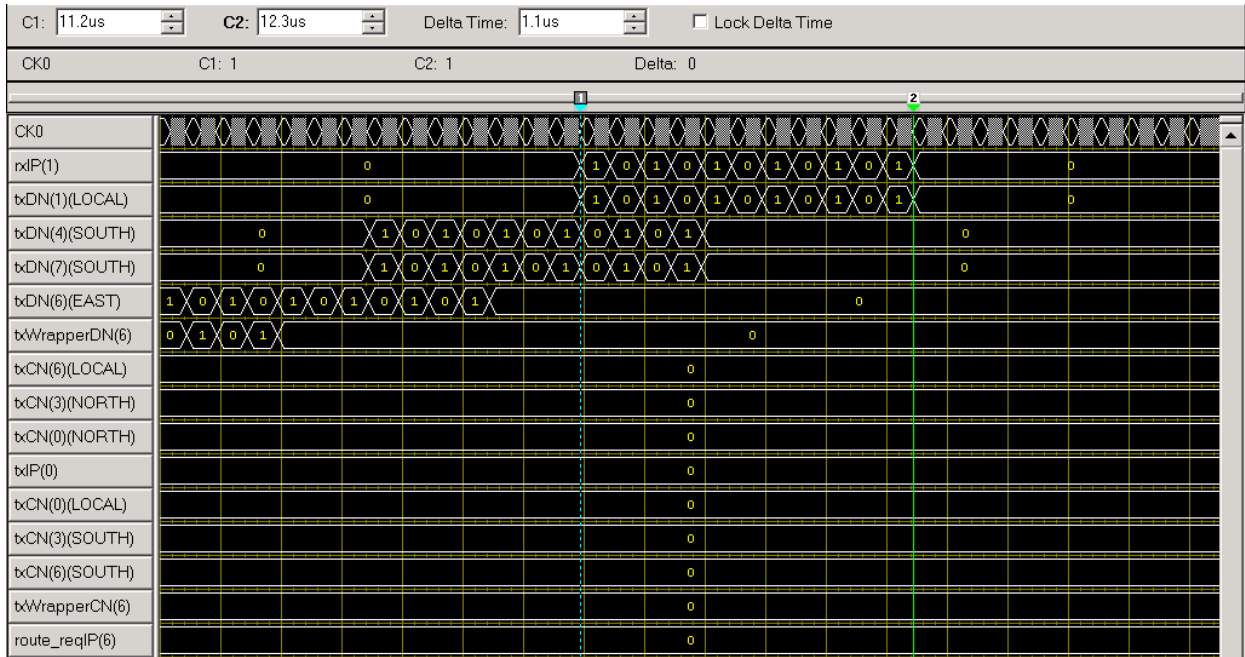


Fig. 5.41: Measurement of the real system - data packet arriving.

In comparison to the PR version without dedicated buffers for the PR router, Fig. 5.42 shows a flit passing the PR router in the version with dedicated buffers. The latency for this increases from basically zero (wires only) to one clock cycle for the buffer (time between the two markers).

Fig. 5.43 finally shows as a comparison the version with a regular router. The latency for the packet passing this router is seven clock cycles, the same as for the other routers.

However, no route establishment and cancelation is needed for this version. In this project only the router is exchanged with a regular one and the wrapper is exchanged with one letting the signals pass directly into the data network. The IP cores still reserve and cancel routes, which is unnecessary for this version. As this version is for comparison only, this can be ignored and the time between the red and the blue marker is not taken into account. The time for the packet to be delivered can be seen as the time between the blue and the green marker, which is $3.9\mu\text{s}$ respectively 39 clock cycles.

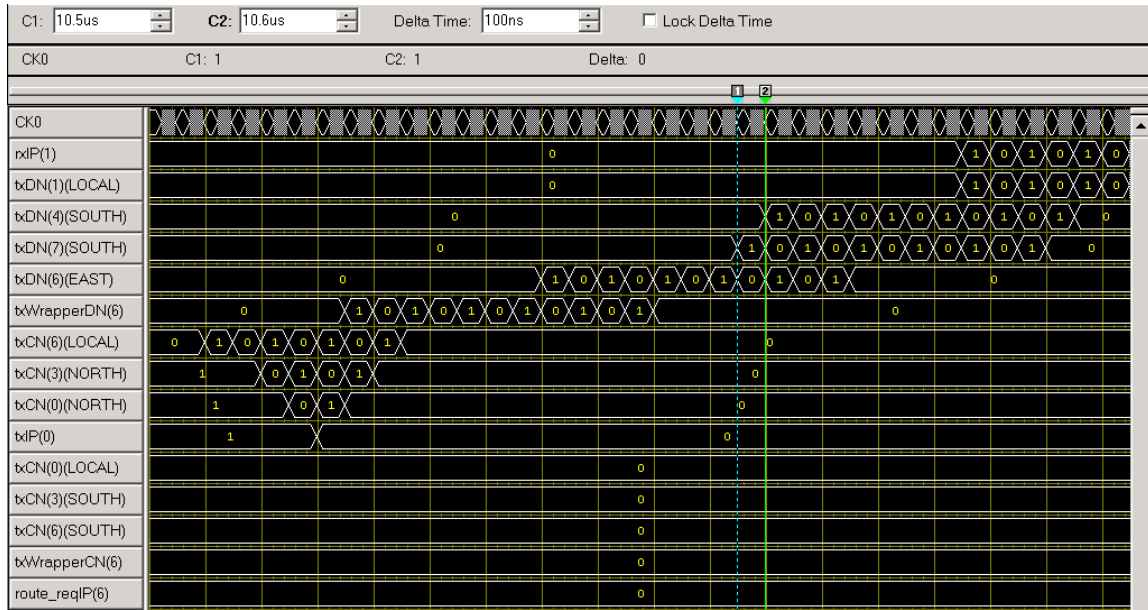


Fig. 5.42: Measurement of the real system - data packet traveling through PR-buffer.

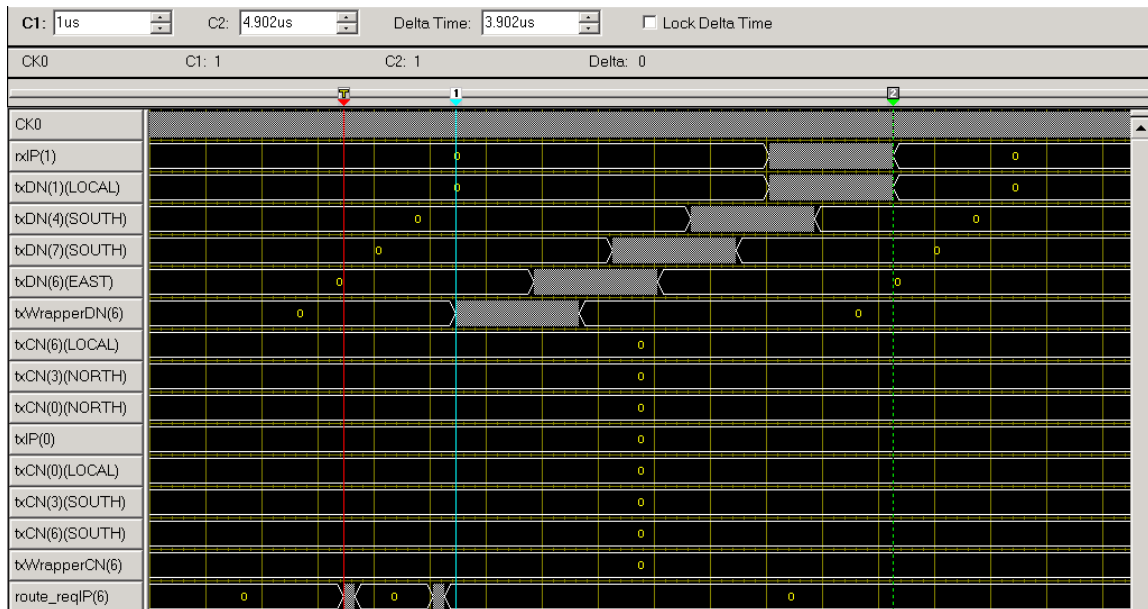


Fig. 5.43: Measurement of the real system - version with regular router.

5.9.2.2 Timing analysis of the PR-process

At last the time needed for the PR-process itself can be measured. The reconfiguration is performed by the attached PC using the JTAG interface. It is the only point that allows

respectively six clock cycles for the non-buffered and buffered version for each router the packet passes, compared with the regular version.

Tab. 5.1: Comparison of latency for the different system versions

	signal	latency PR	latency PR_Buffer	latency regular	comment
route cancelation	route_reqIP(6)				start
	txWrapperCN(6)	2	2		latency of wrapper
	txCN(6)(SOUTH)	6	6		latency of control router
	txCN(3)(SOUTH)	6	6		latency of control router
	txCN(0)(LOCAL)	6	6		latency of control router
	route_reqIP(6) (*)	8	8		time for wrapper to become available
route establishment	txWrapperCN(6)	2	2		latency of wrapper
	txCN(6)(SOUTH)	7	7		latency of control router
	txCN(3)(SOUTH)	6	6		latency of control router
	txCN(0)(LOCAL)	6	6		latency of control router
	txIP(0)	19	19		latency of PR-Master (varies by 120, PR-process not included!)
	txCN(0)(NORTH)	6	6		latency of control router
	txCN(3)(NORTH)	6	6		latency of control router
	txCN(6)(NORTH)	6	6		latency of control router
	txWrapperDN(6)	5	5		latency of wrapper
data transfer	txDN(6)(EAST)	7	7	7	latency of data router
	txDN(7)(SOUTH)	0	1	7	latency of PR-router
	txDN(4)(SOUTH)	7	7	7	latency of data router
	txDN(1)(LOCAL)	7	7	7	latency of data router
	rxIP(1)	11	11	11	time for packet to be received
	Total latency	123	124	39	

5.10 Area analysis

Finally the FPGA utilization of the regular Hermes NoC and the PR Hermes NoC can be compared. For this the utilization in form of FFs and LUTs of the individual parts of the network is compared. This allows an estimate of utilization for an arbitrary size of network and shows which parts use which resources most. For this the modules have been synthesized separately, for best comparison with an address of "0101" and the routers connected to all five neighbors (four other routers and one local core). Border routers may use less resources as they connect to less than five neighbors. This effect however can be neglected for larger networks. Also it is assumed that all data routers are PR routers, not only one as exemplarily shown in this work.

Tab. 5.2 shows the comparison. The regular NoC consists of data routers only. Their number equals the size of the network, for a three-by-three network nine in total. The PR NoC consists of the control routers, the PR-routers, their wrappers and the PR-Master. Their number also equals to the size of the network, except for the PR-Master, which is needed only once for an arbitrary network size.

Tab. 5.2: Comparison of FPGA utilization for regular and PR design

	# needed for network of size n	Flip- Flops	LUTs
Regular			
data router	n	164	761
PR			
control router	n	132	582
PR-router (*)	n	224	224
wrapper	n	64	102
PR-Master	1	189	264

(*) see text for details

With this table the FPGA utilization for a network of arbitrary size can be estimated. Fig. 5.45 and Fig. 5.46 shows the utilization of FFs respectively LUTs for a network of size one to 20. As can be seen the PR design is larger than the regular design for any network size. The expectation was that the curves cross and starting from a certain network size the PR design would be smaller than the regular design. For this the regular router would have needed to be larger than the sum of the PR-router, the control router and their wrappers. The PR-Master does not need to be taken into account as it can be neglected for a larger network. Once the overhead for the PR-Master would have become smaller then the savings from the network, the curves would have crossed, making the PR-design more efficient.

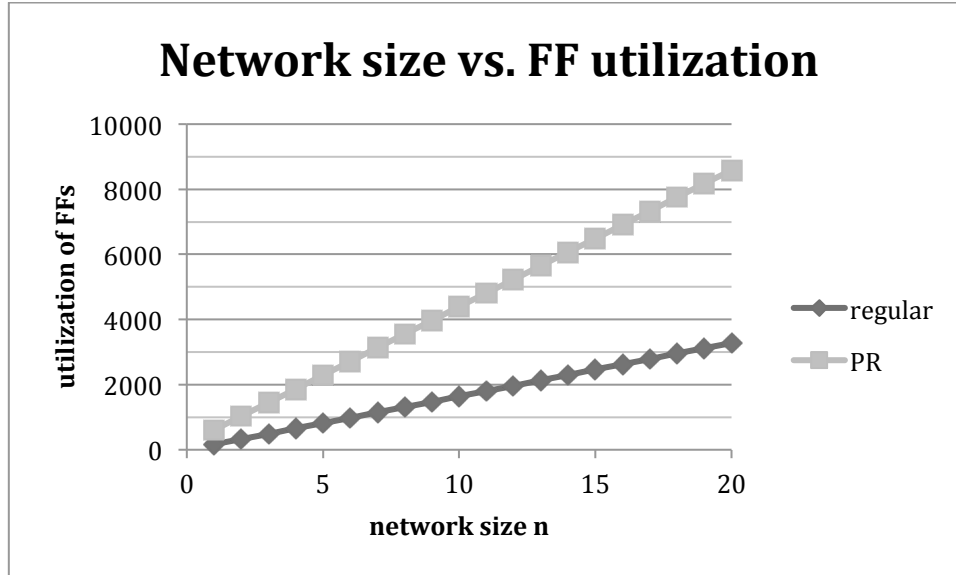


Fig. 5.45: Comparison of the flip-flop utilization in dependence of the network size for a regular and a PR design.

Unfortunately this is not the case here. There are two basic reasons for this. First, the size of the PRR is quite large, occupying 28 CLBs which represent 224 FFs and 224 LUTs. The region cannot be designed smaller as PAR fails otherwise. Even though most of the resources within the PRR are unused, they cannot be occupied by any other part of the design. Therefore they are considered as being used for the PR router. Second, the control router is not much smaller than the data router. Even though the control router

is designed much smaller with regard to data-width and buffer space, the router saves only 20% off FFs and 24% of LUTs when compared to the larger data router. Making this router much smaller, or even substituting the complete control network by a shared bus, could solve this problem. This would hardly affect the performance of the overall system, as the performance of the control network basically does not contribute to it.

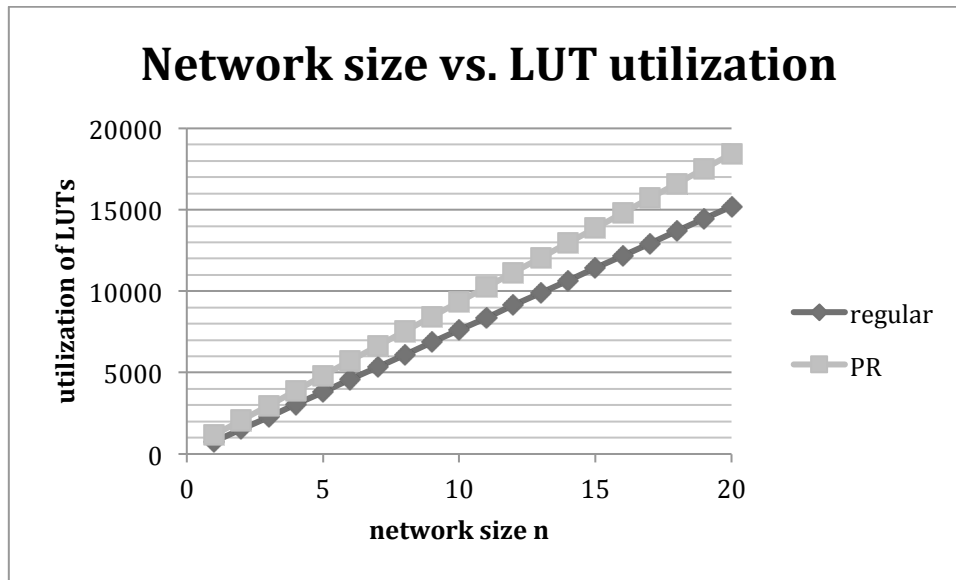


Fig. 5.46: Comparison of the LUT utilization in dependence of the network size for a regular and a PR design.

5.11 Summary

It was shown on this Chapter that partial reconfiguration can be used for changing the routing tables of a NoC. For this reason a NoC was implemented matching the requirements of the EAPR flow, and the design was shown to work correctly on a real FPGA. The EAPR flow demands strong restrictions on the design, especially with regard to hierarchy.

One of the major metrics for a NoC besides bandwidth, which is not affected here, is the latency within the system. Comparing the regular NoC to the buffered PR version, the latency for a packet to travel through the NoC is reduced from seven to only one clock cycle per passed PR-router. However, an initial delay is introduced, as routes now need to be reserved before they can be used. This currently takes about 17.3ms. This time could be reduced to estimated 20 μ s by the use of the ICAP interface, internally reconfiguring the FPGA. The drawback of the initial delay could be tolerated in return to the gain of lower latency for the data transport, especially in systems where long-living connections exist.

In the current demonstration design only one router is partially reconfigurable. To substitute all regular routers with PR-routers, while using the same PRMs for all of them, the PRMs need to be shifted in place. Normally, PRMs include the location where they are placed on the FPGA. Becker et al. [70] show how PRMs can be shifted to a different area, even when the resources of the areas partly differ.

Another important metric is the utilization of the design on the FPGA. The initial assumption was that the PR-version would use less resources than the regular one. This is not the case, and there is no break-even point for a larger NoC that will satisfy this assumption. The main reason for this is because the PR-region requires extra area to place the bus-macros, which leads to an area consumption 3 times higher than expected. This is illustrated on Fig. 5.26, where two columns of CLBs are used by the bus-macros and only one column of CLBs is used by the actual routing logic.

6 Non-Reconfigurable Dual-Layer NoC

As presented on Section 5.11, the reconfigurable data router for the dual-layer NoC required 3 times more area than expected, it was slow to perform the partial reconfiguration and required a centralized configuration controller. In other words, even though partial reconfiguration was theoretically a good idea to configure a runtime data communication between two IP cores without all the cost of completely replicating NoC routers, it has turned out to add the disadvantages just mentioned. As all these drawbacks come from the partial reconfiguration issue, this Chapter proposes a non-reconfigurable dual-layer NoC architecture, while maintaining the idea of low area overhead data routers.

6.1 Proposed Architecture

Similarly as presented on Chapter 5, the dual-layer NoC is composed by a control and a data network as illustrated on Fig. 6.1. The difference is that the centralized configuration controller is no longer required, since the configuration of the data routers is now executed directly by the control routers. The configuration of the data routers is now faster due to the limited amount of logic change required to configure a data router in comparison to perform a partial reconfiguration of all the frames where the data router is located. Borrowing or sharing arbitration and routing logic becomes more advantageous in terms of area as the number of data routers managed by one control router increases.

6.2 Communication Protocol

The best way to show the advantages and disadvantages of such architecture is to first explain an example of how a video can be transferred from an IP core A to an IP core B, while keeping in mind the architecture depicted on Fig. 6.1. Before actually transferring the video through the NoC, IP core A needs to send a ‘establishment packet’ through the control network to IP core B for two distinct reasons. The first is to know if IP core B can accept the video transfer, and the second is to start reserving a communication path through the data network. The control routers probe all packets that are forwarded, and if an ‘establishment packet’ is detected, then the data network is configured to connect the same input port to the same output port used by the ‘establishment packet’. If it happens that the ‘establishment packet’ finds its path blocked in the control network,

the ‘establishment packet’ waits, because the congestion will soon be solved since only small packets are allowed to be transferred by the control network and these packets should not occur too frequently in the control network. If it happens that one input or output port of the data router that is trying to establish a connection is busy, than this will be immediately written inside of the ‘establishment packet’ that is currently being sent to IP core B. As soon as IP core B receives the ‘establishment packet’ from the control network, it will verify if the complete data path was established and if it can receive a video transfer. If so, IP core B will send an ‘acknowledgement packet’ back to IP core A through the control network, otherwise it will send a ‘not acknowledgement packet’. If IP core A receives an ‘acknowledgement packet’, it will immediately start transferring the video through the reserved path on the data network. If IP core A receives a ‘not acknowledgement packet’, then it will send a ‘release packet’ to release the pre-established path.

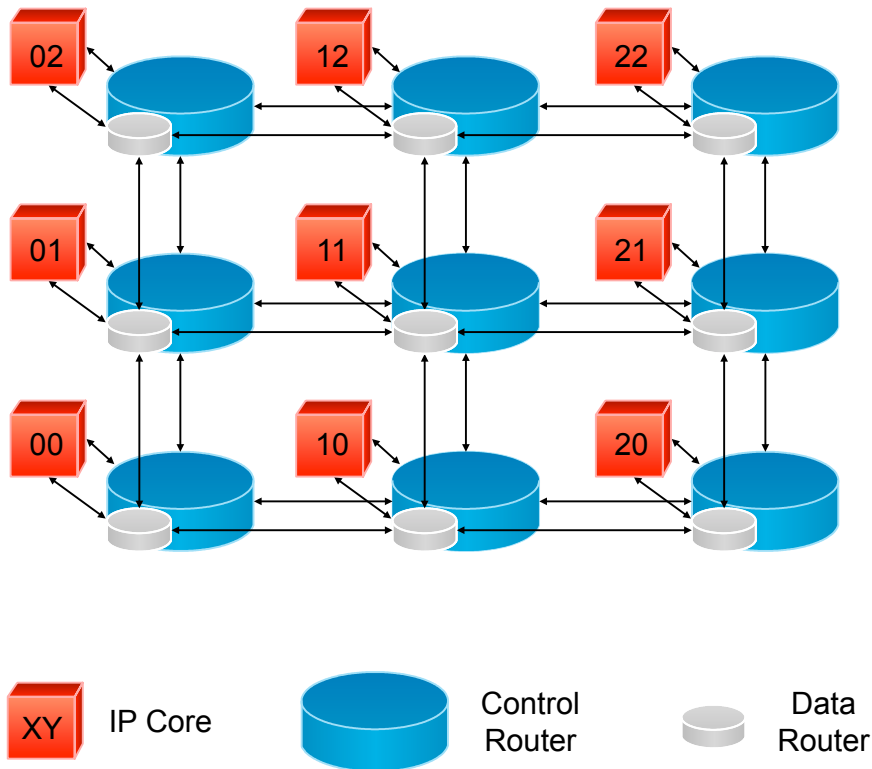


Fig. 6.1: Dual-layer NoC proposed on this work, with one control network and one data network.

6.3 Expected advantages and disadvantages

The only disadvantage found about this dual-layer NoC is that this communication protocol presented on Section 6.2 obliges two packets exchange before any data communication. This is not so problematic because the latency of each control router is 3 clock cycles and the control network is not likely to face congestions because only small control packets should be sent through the control network. If the application running over the dual-layer NoC can really not wait this small latency, the dual-layer NoC could be extended with a small standard packet switched network for data purposes. The advantages of this dual-layer NoC are:

- Quality-of-Service: after the connection is established on the data network, data is transferred in a constant rate;
- Area: data routers contain only one-flit buffers and a crossbar to connect inputs to outputs. No arbitration or routing happens in the data network;
- Scalability: this dual-layer NoC can be extended with more data networks, and the dimensions of each network can also be increased;
- Diversity: different flavors of NoCs can be connected together (e.g. circuit switching, packet switching, networks allowing multicast, ...) in case of a multi-layer NoC is built;
- Priority Packets: packets with high priority could try to reserve more than one data path at a time to reduce the probability of failing to find a data path;
- Extended Bandwidth: data transfers requiring extra bandwidth could try to reserve more than one data path in case of a multi-layer NoC is built;
- Responsiveness: as the control network only exchanges small packets occasionally, a positive or negative response from the destination IP core is always quickly received.

6.4 Implementation

6.4.1 Choice of FPGA and software

The dual-layer NoC was simulated using ModelSim from Mentor Graphics [71] and then tested on a Xilinx Virtex-5 ML507 Embedded Evaluation Platform. The Xilinx Integrated Software Environment (ISE) Design Suite version 10.1 software was used to build the dual-layer NoC for the Virtex-5 FPGA.

6.4.2 Hierarchy of the system

Fig. 6.2 demonstrates the system structure hierarchy. The top level of the system instantiates the NoC and the IP cores. The NoC module is composed by the control and data networks. Each network is composed by routers and links. Control routers have crossbar, buffer and switch control, while data routers have only crossbar and buffer. Every IP core is composed of a wrapper and an IP. The wrapper has a control wrapper and a data wrapper, which have both a sender and a receiver.

6.4.3 Control router

Fig. 6.3 presents a block diagram of the control router. A control router is composed by a switch control (containing an arbiter and a routing engine), up to five buffers, a crossbar and a data router control. Arbitration, routing and crossbar do not require changes from the HERMES NoC used as a starting point for this NoC and presented in detail on Section 2.2.3. The data router control was created from scratch. The buffer required

modifications to detect the control packets mentioned shortly on Section 6.2. The following Sections present the new buffer and the data router control module.

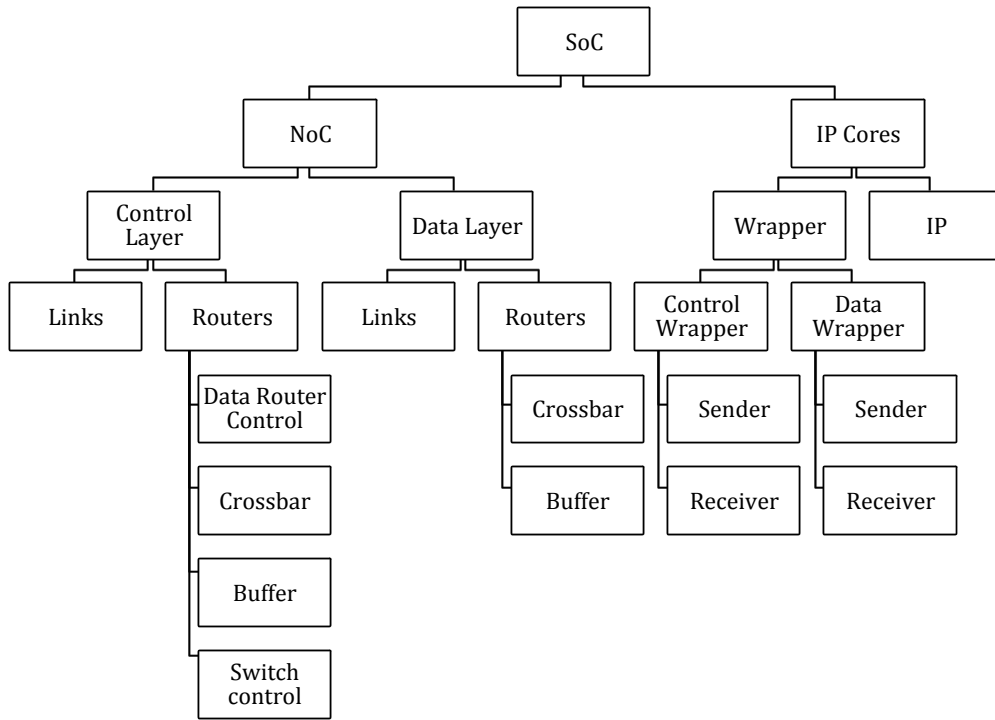


Fig. 6.2: Hierarchy of the system.

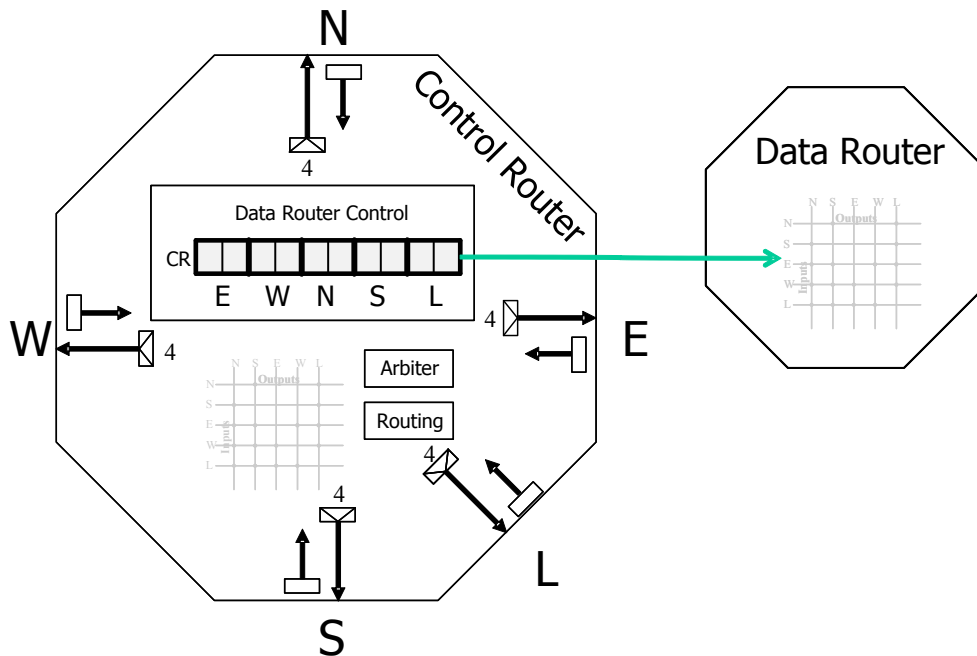


Fig. 6.3: Block diagram of the control router and its connection to the data router.

6.4.3.1 Buffer

The buffer was already not only responsible for temporarily buffering the packets that

were waiting for a congestion to resolve or waiting for arbitration and routing, but also for implementing the flow control between this buffer and the buffer located on the previous router (or the wrapper of the IP core that is sending the packet). Now, for the dual-layer NoC to work, the responsibility of detecting control packets is also passed to the buffer. Such functionality should go to a new module inside the router, but making so would add at least an extra clock cycle delay to the router and increase the area consumption, which is not desirable. The buffer is actually the only place inside of the router where the flits are stored, and therefore was selected to check which control packet was received.

As explained on Section 2.2.3.3 and illustrated on Fig. 2.4, the standard packet format of HERMES NoC requires two flits as header, which are the ‘target flit’ and the ‘size flit’. For the control network to work according to the communication protocol defined on Section 6.2, a third flit called ‘source flit’ and a fourth flit called ‘command flit’ are required. The ‘source flit’ is required by the destination IP core to know to which IP core to reply back. The ‘command flit’ is required by the intermediate control routers to know which operation to perform. Fig. 6.4 presents all control packets currently accepted by the control network.

	1st flit	2nd flit	3rd flit	4th flit
Establishment Packet	Target (XY)	Size (2)	Source (XY)	Command (00)
Establishment Packet Blockage Found	Target (XY)	Size (2)	Source (XY)	Command (10)
Acknowledgement Packet	Target (XY)	Size (2)	Source (XY)	Command (01)
Not Acknowledgement Packet (target busy)	Target (XY)	Size (2)	Source (XY)	Command (02)
Not Acknowledgement Packet (data router busy)	Target (XY)	Size (2)	Source (XY)	Command (03)
Release Packet	Target (XY)	Size (2)	Source (XY)	Command (04)
Release Packet Blockage Found	Target (XY)	Size (2)	Source (XY)	Command (14)

Fig. 6.4: Control packets accepted by the control network.

These control packets exist to deal with three possible outcomes for establishing a data communication: (1) data communication established successfully; (2) data communication not established because the target is busy; (3) data communication not established because at least one data router has one required port which is currently busy. These three possible situations are illustrated in the sequence diagram of Fig. 6.5, where the dashed arrow represents a data transfer over the data network, and the other arrows represent a control packet transfer over the control network. The numbers over the arrows represent the commands specified on Fig. 6.4.

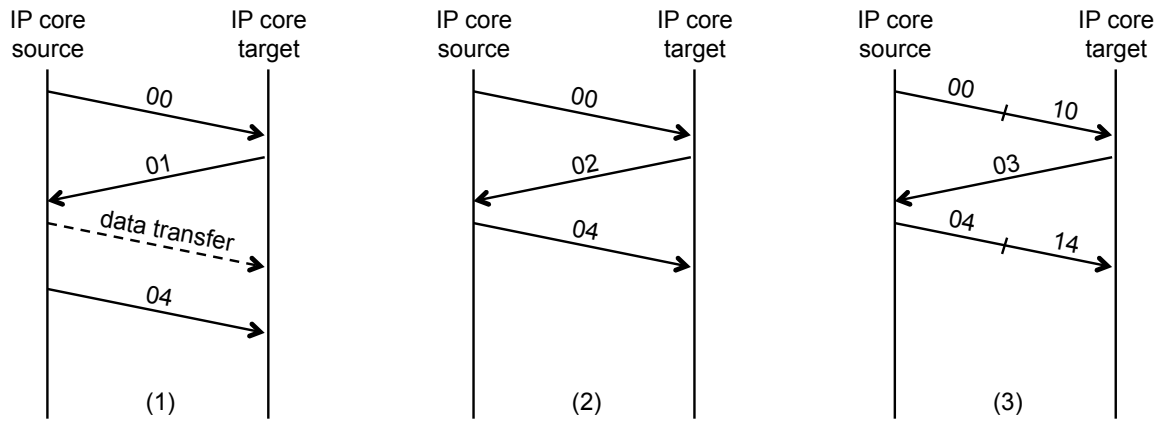


Fig. 6.5: Sequence diagrams of the three possible outcomes for starting a data communication.

Situation 1 describes the successful data transfer from IP core source to target. In this case the 'establishment packet' (command 00) is able to reserve the complete data path between source and target. The target sends an 'acknowledgment packet' (command 01) informing that data can be received. The source then sends all the data to the target and after that it sends a 'release packet' (command 04) to cancel the data path and to inform the target that the data communication has finished.

Situation 2 describes the case where the target IP core is busy and it cannot receive data in the moment. In this case the 'establishment packet' (command 00) is able to reserve the complete data path between source and target. However, the target IP sends a 'not acknowledgement packet (target busy)' (command 02). Then, the IP core is forced to send a 'release packet' (command 04) to cancel the reserved data path without having sent the desired data.

Situation 3 describes the case where a data communication cannot be set because at least one data router has one required port busy. As this case is a little more complex than the previous ones, an example where this situation happens is illustrated on Fig. 6.6. The numbering on the figure refers to different time points explained below.

1. A data communication between IP 00 and IP 01 is already in progress.
2. IP 10 wants to establish a data communication with IP 02, therefore the wrapper of IP 10 sends a 'establishment packet' over the control network following the standard XY routing algorithm. The data router 10 successfully connects the input local port to the west output port.
3. The 'establishment packet' starts to be received on the east buffer of the control router 00. Arbitration and routing occur. East buffer is now connected to the north output port. The first three flits of the 'establishment packet' are forwarded to the output port. When the fourth flit arrives, the control router realizes that the east input of the data router cannot be connected to the north output because the north output is already used by the data communication between IP 00 and IP 01. Then the control router forwards the flit 10 instead the flit 00, transforming the 'establishment packet' on an 'establishment packet blockage found'.
4. The control router 01 receives the 'establishment packet blockage found' and only

forwards towards its target without performing any operation.

5. The wrapper of IP 02 receives the 'establishment packet blockage found'.
6. The wrapper of IP 02 sends a 'not acknowledgement packet (data router busy)' back to IP 10, informing that the data communication was not totally established.
7. The wrapper of IP 10 receives the 'not acknowledgement packet (data router busy)'.
8. The wrapper of IP 10 sends a 'release packet' towards IP 02, to release the partial reserved data path between them. The data connection established on time point 2 between the input local port and the west output port is now released.
9. The 'release packet' starts to be received on the east buffer of the control router 00. Arbitration and routing occur. East buffer is now connected to the north output port. The first three flits of the 'release packet' are forwarded to the output port. When the fourth flit arrives, the control router realizes that the east input of the data router is not connected to any output port. Then the control router forwards the flit 14 instead the flit 04, transforming the 'release packet' on a 'release packet blockage found'.
10. The control router 01 receives the 'release packet blockage found' and only forwards towards its target without performing any operation.
11. The wrapper of IP 02 only receives the 'release packet blockage found' and do not perform any operation.

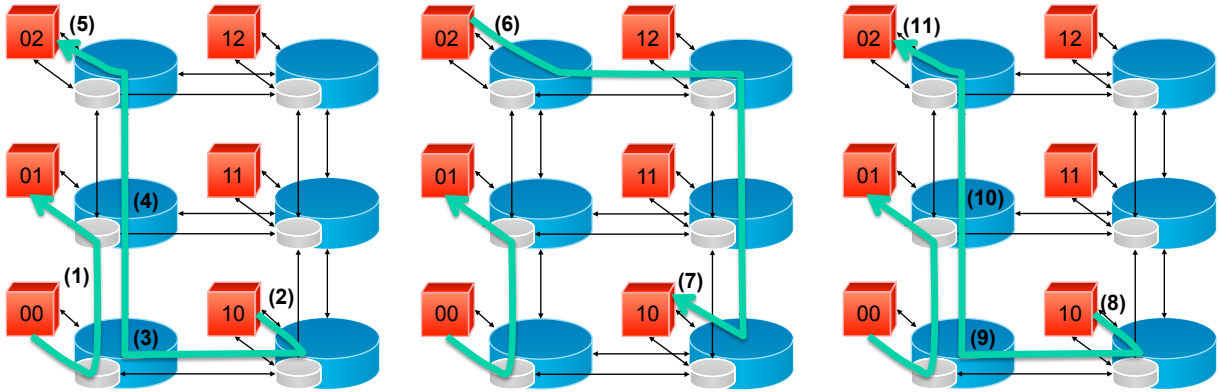


Fig. 6.6: Example of a situation where the data communication is not established because at least one data router has one required port which is currently busy.

This situation illustrated by Fig. 6.6 would not need this complex operation of modifying an ongoing packet if the control router could bufferize the whole control packet and analyze it before forwarding. However, this would require more area to temporarily store the flits of the control packet and it would add extra delay to wait for all flits before forwarding. Besides, the packet switching NoC would no longer be using the wormhole forwarding technique, but the store-and-forward.

6.4.3.2 Data router control

As presented on Fig. 6.3, the data router control is a hardware module from the control router. This module is responsible to set and unset connections from inputs to outputs of the data router attached to it. As soon as a buffer of the control network detects either an 'establishment packet' or a 'release packet', the buffer forwards this information to

the data router control. The data router control has complete access to the routing table of the switch control. With these inputs, the data router control is able to perform one of the following operations:

- If the received packet was an ‘establishment packet’, then the input and output ports are both checked if they are currently available on the data router. If so, the connection between input and output port is established in the data router, and the buffer is acknowledged back. If not, the connection is not established because either the input or the output port is busy, and a not acknowledgement is sent back to the buffer. If the buffer receives an acknowledgement from the data router control, then the buffer forwards the fourth flit with the same information as received (‘establishment packet’ command). If the buffer receives a not acknowledgement from the data router control, then the buffer forwards the fourth flit with the command ‘establishment packet blockage found’.
- If the received packet was a ‘release packet’, then the input and output ports are just marked as free.

It is worth to mention that neither the ‘acknowledgement packet’ nor the ‘not acknowledgement packet’ require any special treatment from the control router and they even can be transferred in a different path from the ‘establishment packet’ or the ‘release packet’ (exactly as presented on time points 6 and 7 of Fig. 6.6), since the XY path from source to target can be different from the XY path from target to source.

As it can be seen on Fig. 6.3, the connection register requires only 10 bits, because each output port (one bold column of the connection register) requires 2 bits to set one of the other 4 input ports which is not the current output direction.

6.4.4 Data router

The routers used here as starting point also come from the HERMES NoC. However, arbitration and routing can be completely removed of the data router, since the output multiplexors are set by the control router. Fig. 6.7 presents an overview of the data router and most of its internal components. The missing components are the acknowledgement signals and their multiplexors, which were removed for the sake of clarity. As presented on Fig. 6.7, the multiplexors are set by the connection register presented on Fig. 6.3. The right side of Fig. 6.7 also presents all connections from inputs to outputs, passing through 1-position buffer and 4-to-1 multiplexors.

6.5 System-on-Chip

This Section presents in more detail all hardware blocks used on the SoC. This includes the test core, the 3x3 dual-layer NoC and IP-network interface.

6.5.1 Test core

The IP cores connected to the dual-layer NoC are simple modules responsible to send

and receive data. The data sent is solely for testing purposes, which mimic the three situations presented on Fig. 6.5. More information about them will be presented on the simulation and prototyping Sections.

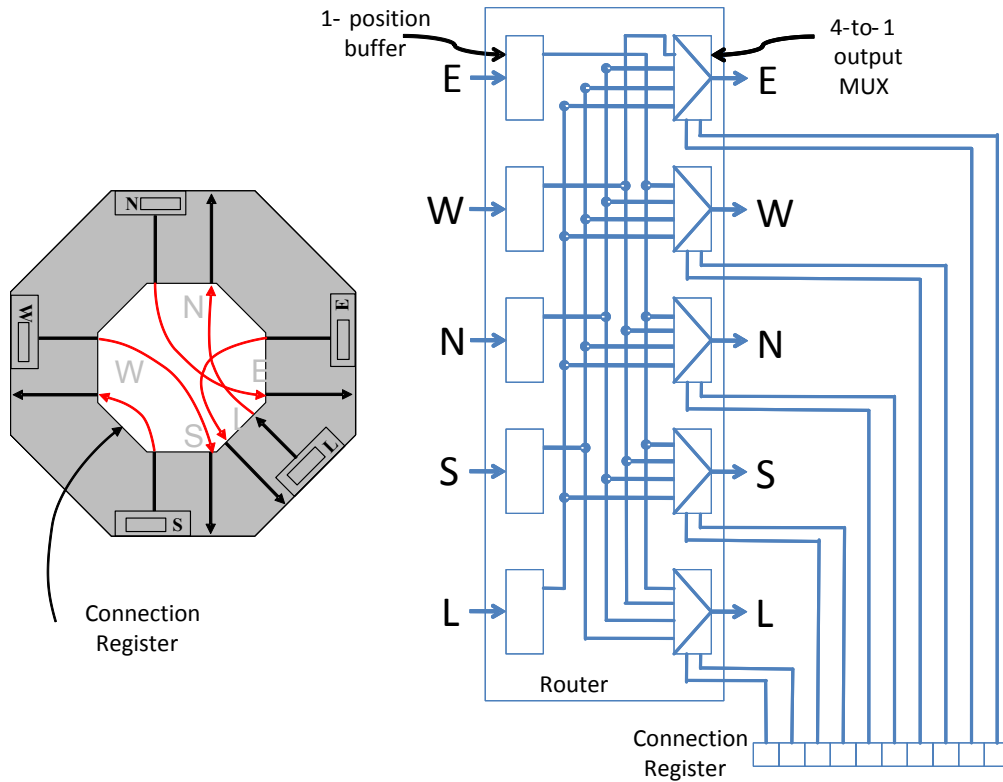


Fig. 6.7: Block diagram of the data router and most of its components.

One important reminder for using the dual-layer NoC at maximum possible speed is to connect processors with Direct Memory Access (DMA) or dedicated hardware blocks that are capable of operating on a frequency similar to the dual-layer NoC. Otherwise the data network paths will be reserved for a period of time longer than required and delaying other traffics to use these same reserved ports of the data routers.

6.5.2 Dual-layer NoC

The dual-layer NoC is divided on control network and data network. Each network is only responsible to instantiate and interconnect its own routers. The control network must contain the same number of routers as the data routers. The following Sections present the control and data routers.

6.5.2.1 Control router

While Section 6.4.3 has presented an overview of the control router, this Section presents it in detail. Fig. 6.8 presents the block diagram of the control router and most of its internal modules. As illustrated, the control router is connected only to the control network. The control router has no logic of its own, being responsible to only interconnect up to five buffers, the data router control, the switch control and the crossbar.

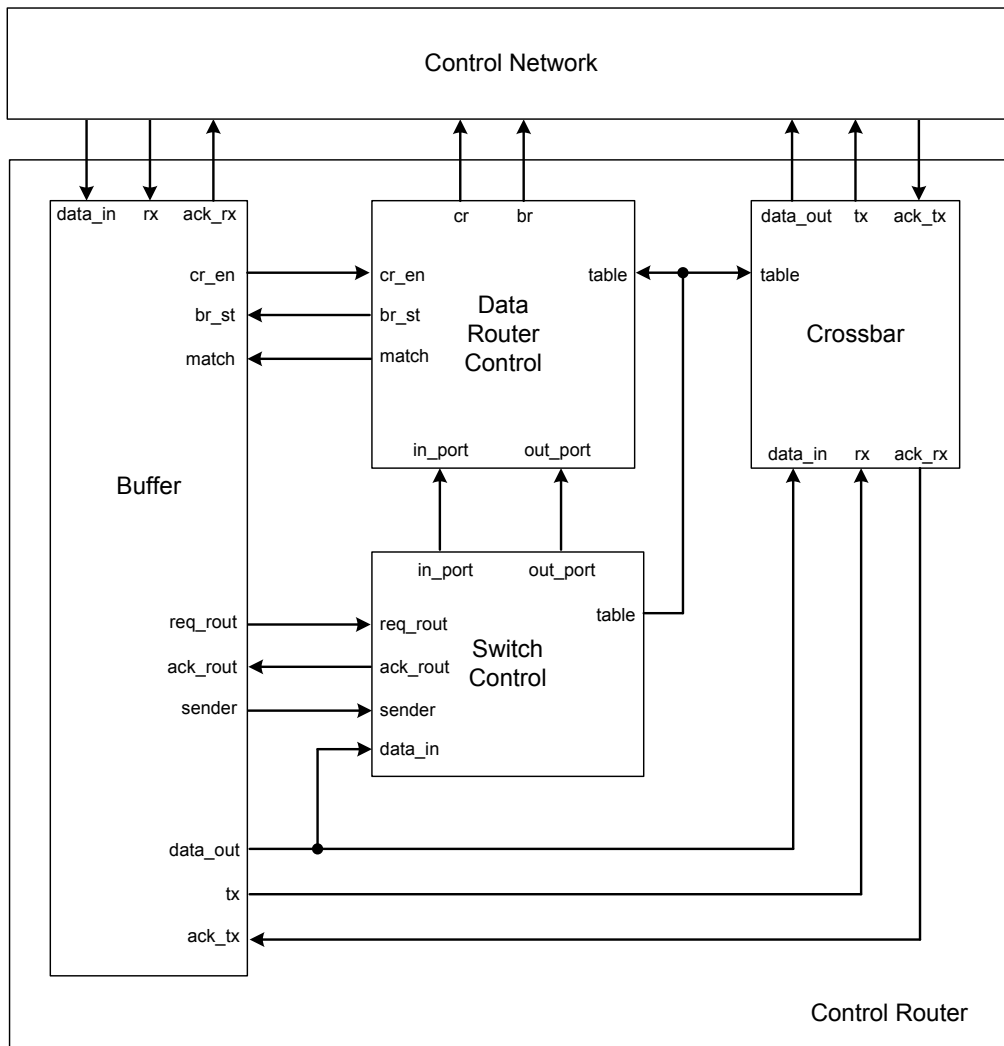


Fig. 6.8: Block diagram of the control router and its internal modules.

As soon as the first flit of a new packet is received on the buffer by the *data_in*, *rx* and *ack_rx* signals as explained on Section 2.2.3, the buffer requests routing for the switch control by asserting the *req_rout* signal high. Together with that, the buffer also sends the first flit of the packet (which contains the target address of the packet) by the *data_out* signal and the information of which input buffer requested routing is sent by the *sender* signal. After executing the round-robin arbitration to choose an input port and executing the routing algorithm XY to get the required output port, the routing *table* is written and the acknowledgement is sent back to the buffer by the *ack_rout* signal. The switch control informs the input and output ports of this last routed packet header by the *in_port* and *out_port* signals to the data router control. The data router control informs the buffer by asserting the *br_st* signal high if either the *in_port* or the *out_port* is used by the data router. When the fourth flit of the packet (which is the command to be executed by the packet) is read by the buffer, three important situations may occur: 1) it is an 'establishment packet' and it will assert the *cr_en* signal high only if *br_st* is asserted low, thus establishing the connection between the *in_port* to the *out_port* on the data router; 2) it is a 'release packet' and it will assert the *cr_en* signal low only if *br_st* is asserted high and the signal *match* is asserted high, confirming that *in_port* is connected

to *out_port* on *table* and thus closing this connection on the data router; 3) it is any other type of packet presented on Fig. 6.4, which will not require any communication between the buffer and the data router and it will perform as explained on Section 6.4.3. If the data router control modifies the data router, this means new values written to the *cr* and *br* signals that directly control the data router as illustrated on Fig. 6.7. While the *cr* signal is the connection register presented on Fig. 6.7, the *br* signal is a busy register that contains 5 bits and each bit is used to inform which of the 5 output multiplexors are currently used. Finally, as soon as the first flit of the control packet is routed to an output port by the switch control, the flits are sent to another router or IP core through the *data_out*, *tx* and *ack_tx* signals as explained on Section 2.2.3.

6.5.2.2 Data router

Fig. 6.9 presents the block diagram of the data router and most of its internal modules. As illustrated, the control router is connected only to the data network. The control router has no logic of its own, being responsible to only interconnect up to five buffers and the crossbar.

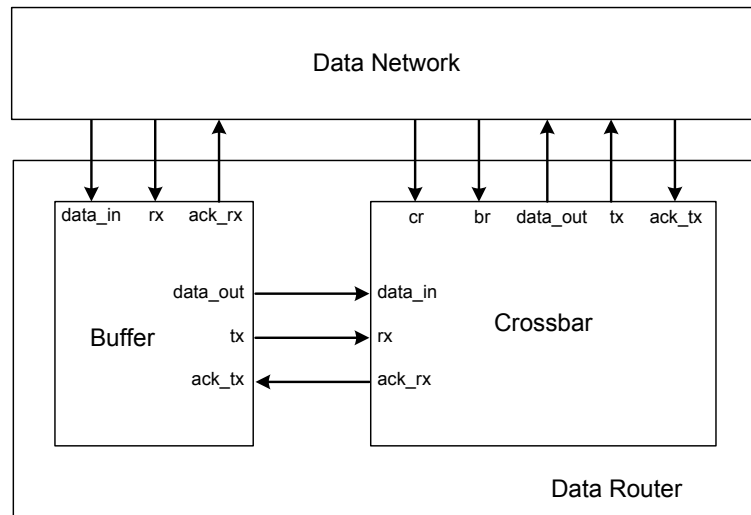


Fig. 6.9: Block diagram of the data router and its internal modules.

Once all data routers between source to target of the communication were configured by the *cr* and *br* signals as explained on Section 6.5.2.1, the data router has only to perform the flow control through the signals *data_in*, *rx* and *ack_rx* and to store flits temporarily on the buffer. The buffer contains only one flit position, and it is used to maintain the pipeline behavior of the NoC. As soon as the flit on the buffer of the next router is free, which is informed by the *ack_tx* signal of the neighbor router, the buffer is able to send another flit through the *data_out* and *tx* signals. The crossbar, as it was already configured by the control router and it connects an input buffer to the right output port, performs no further action.

6.5.3 Network interface

The network interface is responsible to connect the IP core to both the control network

and the data network. The network interface is further subdivided on control wrapper and data wrapper, as presented on Fig. 6.10.

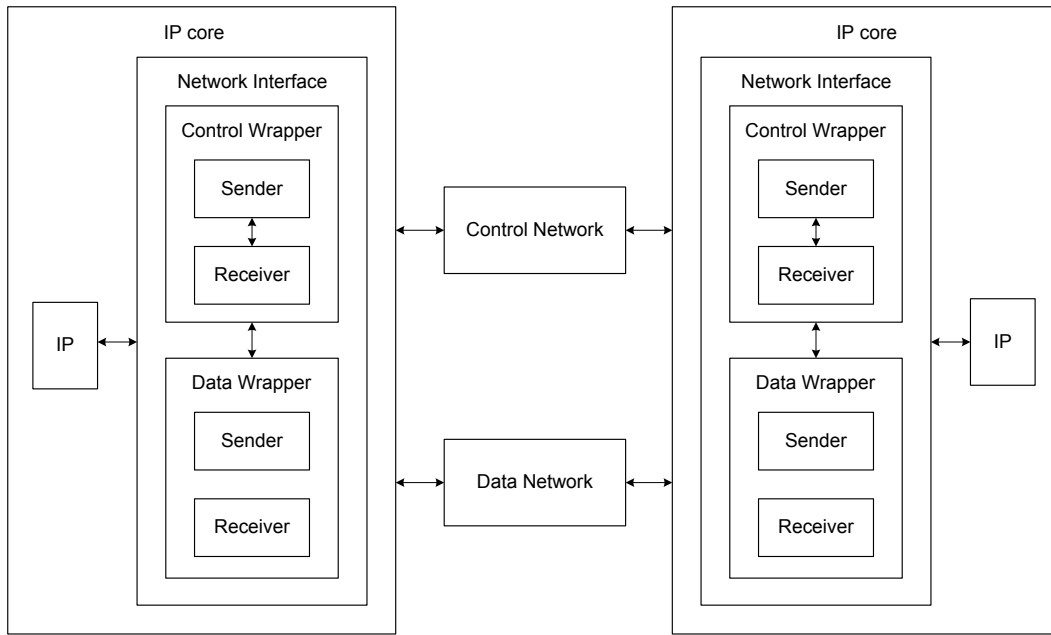


Fig. 6.10: Simplified block diagram between the communication of two IP cores.

6.5.3.1 Control wrapper sender

While Fig. 6.11 depicts the block diagram of the control wrapper sender, Fig. 6.12 illustrates its finite state machine (FSM). The control wrapper sender performs the following functions.

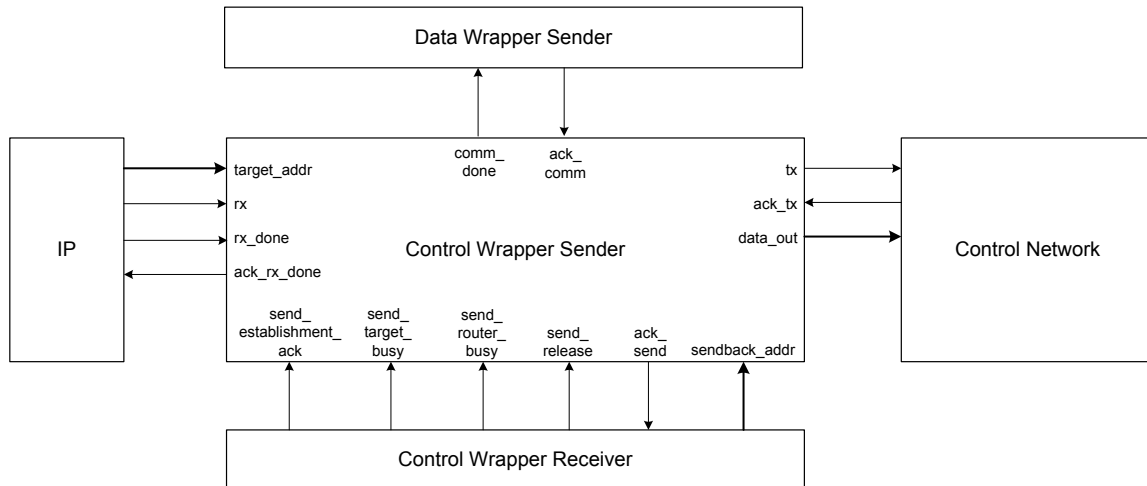


Fig. 6.11: Block diagram of the control wrapper sender.

- When an IP core source wants to send data to an IP core target, the IP core first asserts the *rx* signal high and informs the target IP address by the *target_addr* signal. Then the *send_establishment* signal (not presented on Fig. 6.11) is asserted high and the FSM goes to the *flit* state. In this state the standard handshake flow control of HERMES explained on Section 2.2.3.3 is used to send flit by flit the 'establishment packet' presented on Fig. 6.4. After that, the FSM goes back to the

idle state.

- After the control wrapper receiver of the target IP has received the ‘establishment packet’, the IP core target may or may not be ready to establish a data communication with the source IP.
- If the target IP is ready to establish a data communication, then the control wrapper receiver informs this to the control wrapper sender by asserting the *send_establishment_ack* signal high and informing the address of the IP core source by the *sendback_addr* signal. The control wrapper sender confirms that back to the control wrapper receiver by asserting the *ack_send* signal high. After that the control wrapper sender goes to *flit* state and use the native standard flow control of HERMES explained on Section 2.2.3.3 to send flit by flit the ‘acknowledgement packet’ presented on Fig. 6.4. After that the FSM goes back to the *idle* state. When the control wrapper receiver of the IP core source receives the ‘acknowledgement packet’, the data is transferred through the data network.
- If the target IP is not ready to establish a data communication, then the control wrapper receiver informs this to the control wrapper sender by asserting the *send_target_busy* signal high and informing the address of the IP core source by the *sendback_addr* signal. The control wrapper sender confirms that back to the control wrapper receiver by asserting the *ack_send* signal high. After that the control wrapper sender goes to *flit* state and use the native standard flow control of HERMES explained on Section 2.2.3.3 to send flit by flit the ‘not acknowledgement packet (target busy)’ presented on Fig. 6.4. After that the FSM goes back to the *idle* state. When the control wrapper receiver of the IP core source receives the ‘not acknowledgement packet (target busy)’, it will inform the IP core source that the target is busy.
- However, if the control wrapper receiver of the target IP has received an ‘establishment packet blockage found’, it means that during the transmission of the ‘establishment packet’ one required port of a data router was used and the data communication was partially established. Therefore, the control wrapper receiver informs that to the control wrapper sender by asserting the *send_router_busy* signal high and informing the address of the IP core source by the *sendback_addr* signal. The control wrapper sender confirms that back to the control wrapper receiver by asserting the *ack_send* signal high. After that the control wrapper sender goes to *flit* state and use the native standard flow control of HERMES explained on Section 2.2.3.3 to send flit by flit the ‘not acknowledgement packet (router busy)’ presented on Fig. 6.4. After that the FSM goes back to the *idle* state. When the control wrapper receiver of the IP core source receives the ‘not acknowledgement packet (router busy)’, it will inform the IP core source that a data path to perform the communication could not be established at this time.
- When the control wrapper sender is in *idle* state, it checks every clock cycle if either the *rx_done* or the *send_release* signal. If so, it means that the data path reserved by the IP core source where this control wrapper sender is connected, should be terminated. While the *send_release* signal is asserted high by the control wrapper

receiver due to the problems explained above (i.e. the target IP core is busy or a required data router is busy), the *rx_done* signal is asserted high by the IP core source as soon as it has finished to communicate with the IP core target. The control wrapper sender confirms back to the IP core by asserting the *ack_rx_done* signal high if the *rx_done* signal was asserted high, or it confirms back by asserting the *ack_send* signal high if the *send_release* was asserted high. Also, if the IP core source has asserted high the *rx_done*, the control wrapper sender asserts the *comm_done* signal high to inform to the data wrapper that the communication is over. The IP core source acknowledges back through the *ack_comm* signal. In any case the FSM jumps to the *flit* state. The standard flow control of HERMES explained on Section 2.2.3.3 is used to send flit by flit the 'release packet' presented on Fig. 6.4. After that the FSM goes back to the *idle* state and the reserved data path is released.

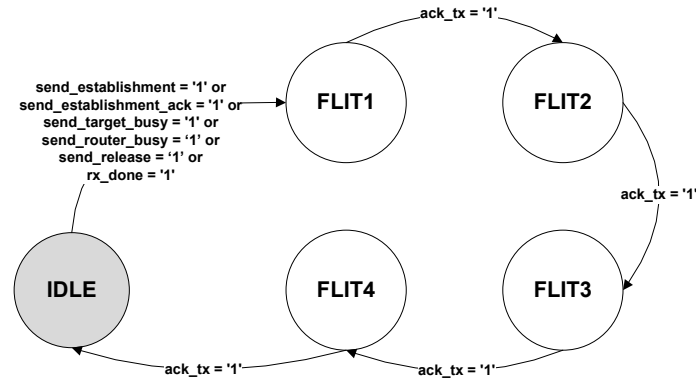


Fig. 6.12: FSM of the control wrapper of the sender.

6.5.3.2 Control wrapper of the receiver

While Fig. 6.13 depicts the block diagram of the control wrapper receiver, Fig. 6.14 illustrates its finite state machine (FSM). The control wrapper receiver starts by receiving one of the control packets presented on Fig. 6.4. The four flits that compose these packets are received according to the standard handshake flow control of HERMES explained on Section 2.2.3.3, and for each flit received the FSM advances to the next state. When the last flit is received in the state *flit4*, then one of the following commands may be recognized:

- 'establishment packet': this control packet is received by the control wrapper receiver of the target IP core when an IP core source wants to establish data communication. As soon as it arrives, the control wrapper receiver asserts the *send_establishment_ack* signal high if the IP core target is free to receive a new data communication (*busy_IP* signal is zero). If the target IP cannot receive a new data communication (*busy_IP* is equal to one), then *send_establishment_nack* is asserted high. After the control wrapper sender has received one of these assertions, it will assert the *ack_send* signal high and the FSM of the control wrapper receiver goes back to *idle* state.
- 'establishment packet failed': this control packet is received by the control wrapper

receiver of the target IP core when an IP core source wants to establish data communication, but a data path through the data network was not able to be fully reserved. As soon as it arrives, the control wrapper receiver asserts the *send_router_busy* signal high. After the control wrapper sender has received this assertion, it will assert the *ack_send* signal high and the FSM of the control wrapper receiver goes back to *idle* state.

- ‘acknowledgement packet’: this control packet is received by the control wrapper receiver of the source IP core when the IP core target accepts to establish a data communication. As soon as it arrives, the control wrapper receiver asserts the *establishment_done* signal high to inform the IP core that it can start the data transfer through the data network. Next, the IP core asserts the *establishment_done_ack* signal high and the FSM of the control wrapper receiver jumps to *idle* state.
- ‘not acknowledgement packet (target busy)’: this control packet is received by the control wrapper receiver of the source IP core when the IP core target is busy and does not accept to establish a data communication. As soon as it arrives, the control wrapper receiver asserts the *target_busy* signal high to inform the IP core source that the data path could not be established at this time. How the IP core will deal with this problem depends on its own implementation. Also, the control wrapper receiver asserts the *send_release* signal high to inform to the control wrapper sender that a ‘release packet’ should be sent to disconnect the reserved path on the control network.
- ‘not acknowledgement packet (data router busy)’: this control packet is received by the control wrapper receiver of the source IP core when the IP core target has received an ‘establishment packet failed’, meaning that a data communication was not fully established. As soon as the ‘not acknowledgement packet (data router busy)’ is recognized, the control wrapper receiver asserts the *router_busy* signal high to inform the IP core source that the data path could not be established at this time. How the IP core will deal with this problem depends on its own implementation. Also, the control wrapper receiver asserts the *send_release* signal high to inform to the control wrapper sender that a ‘release packet’ should be sent to disconnect the partially reserved path on the control network.
- ‘release packet’: this control packet is received by the control wrapper receiver of the target IP core when the IP core source wants to close the communication with the target IP core. As soon as the ‘release packet’ is recognized, the control wrapper receiver asserts the *end_of_data* signal high to inform the IP core target that the IP core source has finished to send data. The IP core target acknowledges back by asserting the *ack_end_of_data* signal high.
- ‘release packet blockage found’: this control packet is initially sent as a ‘release packet’ by the wrapper control sender of the source IP core and it disconnects the data routers of a partially pre-established path. When the ‘release packet’ arrives to the control router which does not connect the input of the data router to the correct output towards to the target IP core, then the control router transforms this packet into a ‘release packet blockage found’. This transformation happens because when

the data router realizes that the 'release packet' should not disconnect the neighbor routers, part of this packet is already in the next router, and the only thing that can be done is to change the command of this control packet. So, when the 'release packet blockage found' is forwarded by the other control routers in the path and finally received by the control wrapper receiver of the target IP core, it performs no operation.

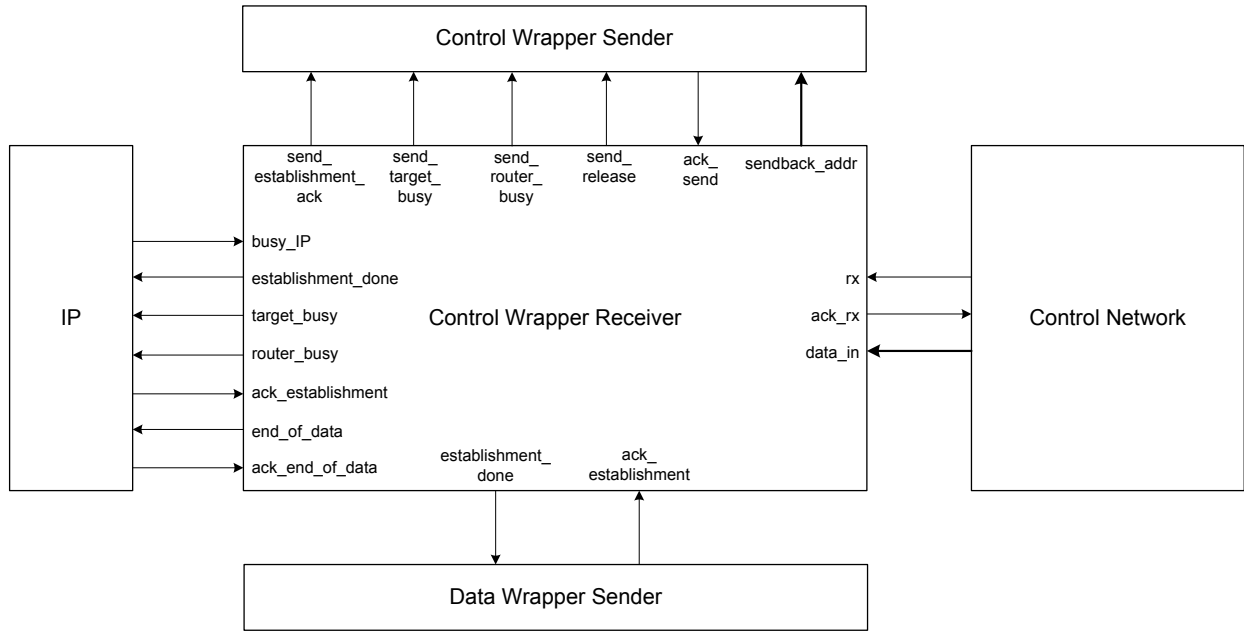


Fig. 6.13: Block diagram of the control wrapper receiver.

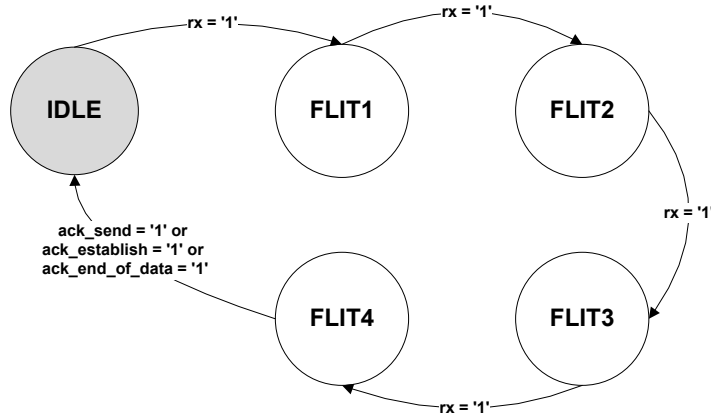


Fig. 6.14: FSM of the control wrapper of the receiver.

6.5.3.3 Data wrapper of the sender

Fig. 6.15 depicts the block diagram of the data wrapper sender. As soon as the *establishment_done* signal is asserted high, it means that the 'acknowledgement packet' was received by the control wrapper receiver and the data path was created successfully between the source and target IP cores. The *data_in*, *rx* and *ack_rx* are then respectively connected to the *data_out*, *tx* and *ack_tx* signals and the data wrapper sender acknowledges back the control wrapper receiver through the *ack_establishment* signal. After that, the IP core source is able to send all desired communication through the data

network flit by flit. When the IP core source has finished to communicate with the IP core target, the control wrapper sender asserts high the *comm_done* signal and the data wrapper sender disconnects the IP core from the data network and acknowledges back the control wrapper sender through the *ack_comm* signal.

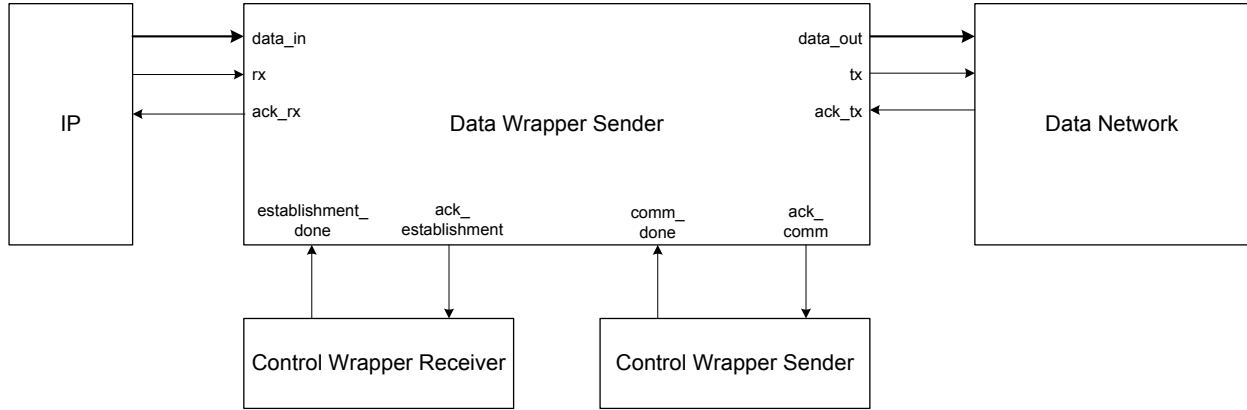


Fig. 6.15: Block diagram of the data wrapper sender.

6.5.3.4 Data wrapper of the receiver

Fig. 6.16 depicts the block diagram of the data wrapper receiver. The data wrapper receiver only receives the flits from the data network and forwards them to the target IP core.

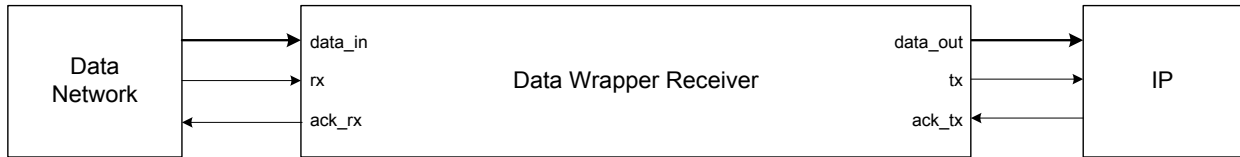


Fig. 6.16: Block diagram of the data wrapper receiver.

6.6 Simulation

This Section focuses on the simulation of the control and data networks. All the experiment data used on this Section is recorded from ModelSim simulations. The clock period used on the simulation is 10ns. The following Sections 6.6.1, 6.6.2 and 6.6.3 have a direct correspondence with the 3 situations illustrated on Fig. 6.5.

6.6.1 Data communication established successfully

Fig. 6.17 presents the simulation results of the IP core 00 performing a successful data communication to IP core 22. The main steps highlighted on the simulation are explained below.

1. Control wrapper sender of IP core 00 sends an 'establishment packet' to control router 00 addressed to IP core 22.
2. Control router 00 starts retransmitting the 'establishment packet' to control router 10 three clock cycles after receiving it. The local input port is connected to the east output port on the data router 00.

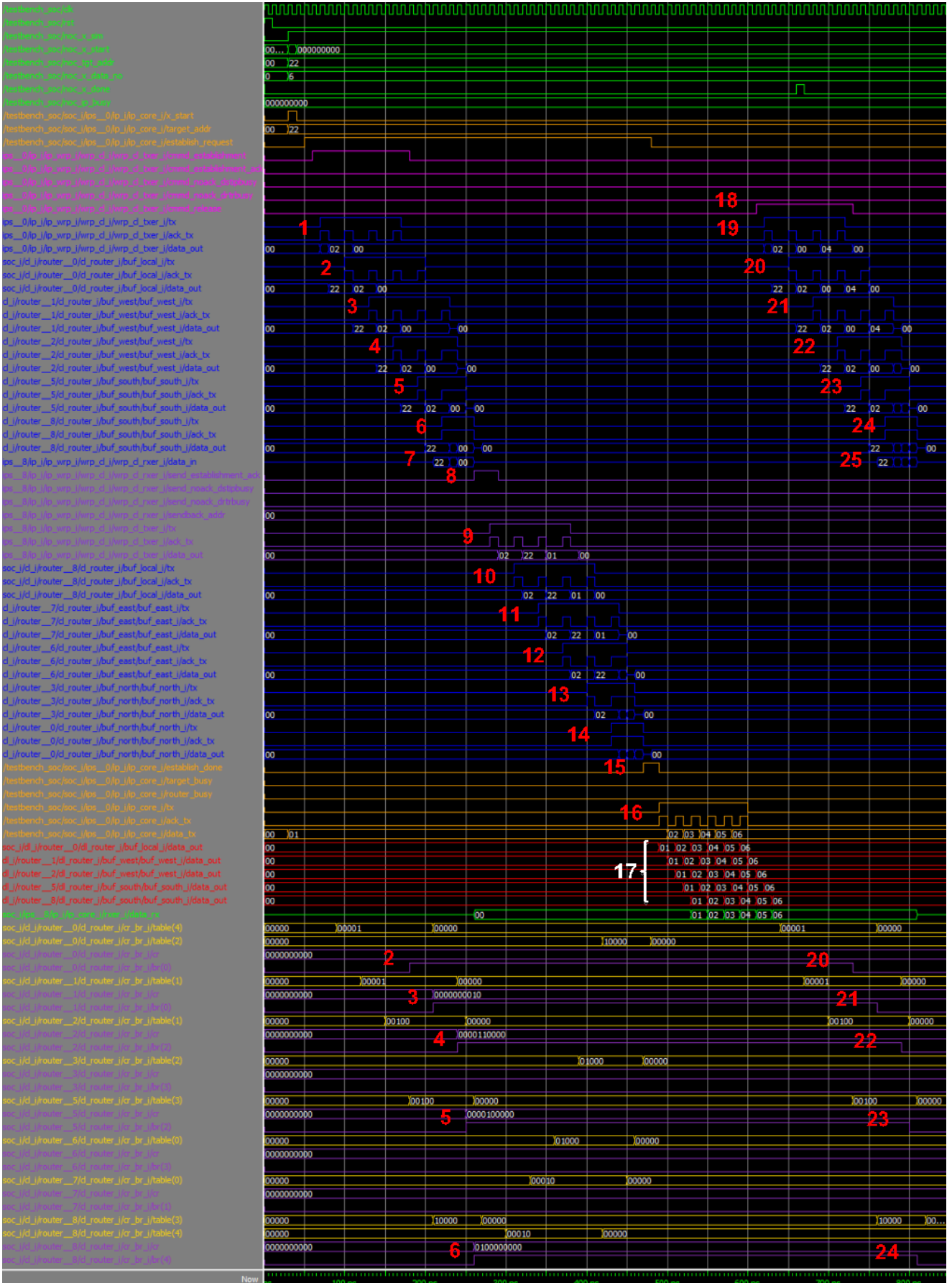


Fig. 6.17: Simulation of a successful data communication.

3. Control router 10 retransmits the packet to control router 20. The west input port is connected to the east output port on the data router 10.
4. Control router 20 retransmits the packet to control router 21. The west input port is connected to the north output port on the data router 20.
5. Control router 21 retransmits the packet to control router 22. The south input port is connected to the north output port on the data router 21.
6. Control router 22 retransmits the packet to IP core 22. The south input port is connected to the local output port on the data router 22.
7. The control wrapper receiver of the IP core 22 starts receiving the 'establishment packet'.
8. As soon as the control wrapper receiver recognizes the 'establishment packet' after receiving the fourth flit of the packet, it asserts the *send_establishment_ack* signal high if the IP core is free to receive a communication (the *busy_IP* signal is not present on the figure).
9. The control wrapper sender starts sending an 'acknowledgement packet' to IP core 00, first transmitting it to router 22.
10. Control router 22 retransmits the packet to control router 12.
11. Control router 12 retransmits the packet to control router 02.
12. Control router 02 retransmits the packet to control router 01.
13. Control router 01 retransmits the packet to control router 00.
14. Control router 00 retransmits the packet to IP core 00.
15. As soon as the control wrapper receiver recognizes the 'acknowledgement packet' after receiving the fourth flit of the packet, it asserts the *establishment_done* signal high.
16. The IP core starts sending the data communication to the data network.
17. Each data router is able to forward a flit in 1 clock cycle. No network congestion exists here.
18. The control wrapper sender is informed by IP core 00 that all data was sent.
19. Control wrapper sender starts transmitting the 'release packet' to control router 00 addressed to control router 22.
20. Control router 00 starts retransmitting the 'release packet' to control router 10. The local input port is disconnected of the east output port on the data router 00.
21. Control router 10 retransmits the packet to control router 20. The west input port is disconnected of the east output port on the data router 10.
22. Control router 20 retransmits the packet to control router 21. The west input port is disconnected of the north output port on the data router 20.
23. Control router 21 retransmits the packet to control router 22. The south input port is disconnected of the north output port on the data router 21.
24. Control router 22 retransmits the packet to IP core 22. The south input port is disconnected of the local output port on the data router 22.
25. The control wrapper receiver of the IP core 22 receives the 'release packet'.

6.6.2 Data communication not established - target busy

Fig. 6.18 presents the simulation results of an attempt of IP core 00 to establish a data

communication to IP core 22, but the data communication cannot be established because IP core 22 is currently busy. The main steps highlighted on the simulation are explained below.

1. Control wrapper sender of IP core 00 sends an 'establishment packet' to control router 00 addressed to IP core 22.
2. Control router 00 starts retransmitting the 'establishment packet' to control router 10 three clock cycles after receiving it. The local input port is connected to the east output port on the data router 00.
3. Control router 10 retransmits the packet to control router 20. The west input port is connected to the east output port on the data router 10.
4. Control router 20 retransmits the packet to control router 21. The west input port is connected to the north output port on the data router 20.
5. Control router 21 retransmits the packet to control router 22. The south input port is connected to the north output port on the data router 21.
6. Control router 22 retransmits the packet to IP core 22. The south input port is connected to the local output port on the data router 22.
7. The control wrapper receiver of the IP core 22 starts receiving the 'establishment packet'.
8. As soon as the control wrapper receiver recognizes the 'establishment packet' after receiving the fourth flit of the packet, it asserts the *send_target_busy* signal high because the IP core is currently not free to receive a communication (the *busy_IP* signal is not present on the figure).
9. The control wrapper sender starts sending a 'not acknowledgement packet (target busy)' to IP core 00, first transmitting it to router 22.
10. Control router 22 retransmits the packet to control router 12.
11. Control router 12 retransmits the packet to control router 02.
12. Control router 02 retransmits the packet to control router 01.
13. Control router 01 retransmits the packet to control router 00.
14. Control router 00 retransmits the packet to IP core 00.
15. As soon as the control wrapper receiver recognizes the 'not acknowledgement packet (target busy)' after receiving the fourth flit of the packet, it asserts the *target_busy* signal high to the IP core 00.
16. The control wrapper sender transmits the 'release packet' to control router 00 addressed to control router 22.
17. Control router 00 starts retransmitting the 'release packet' to control router 10. The local input port is disconnected of the east output port on the data router 00.
18. Control router 10 retransmits the packet to control router 20. The west input port is disconnected of the east output port on the data router 10.
19. Control router 20 retransmits the packet to control router 21. The west input port is disconnected of the north output port on the data router 20.
20. Control router 21 retransmits the packet to control router 22. The south input port is disconnected of the north output port on the data router 21.
21. Control router 22 retransmits the packet to IP core 22. The south input port is disconnected of the local output port on the data router 22.
22. The control wrapper receiver of the IP core 22 receives the 'release packet'.

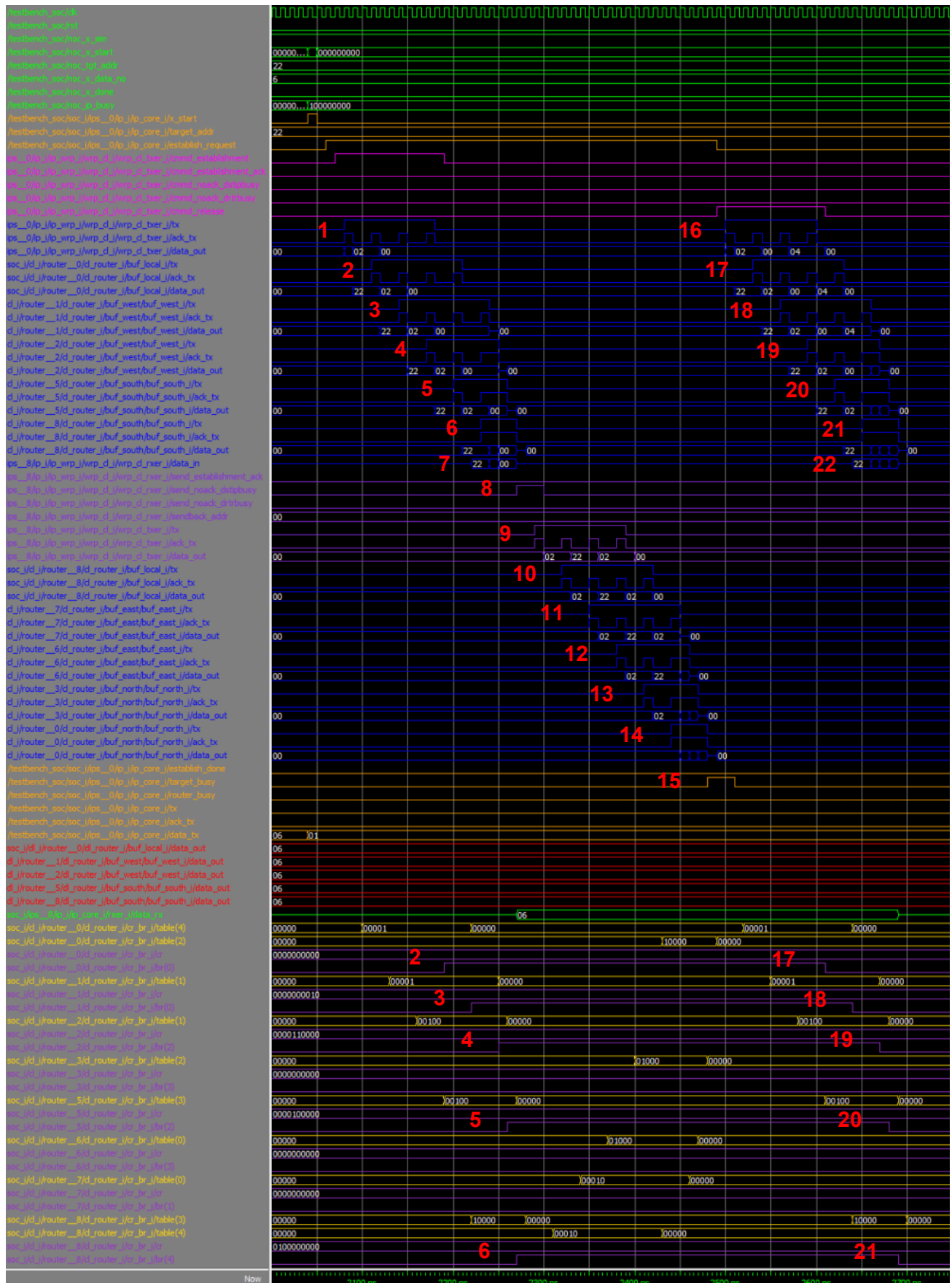


Fig. 6.18: Simulation of a situation where a data communication was not established because the target IP core is busy.

6.6.3 Data communication not established - router busy

Fig. 6.19 and Fig. 6.20 present the simulation results of an attempt of IP core 01 to establish a data communication to IP core 22, but the data communication cannot be established because IP core 22 is already communicating with IP core 00. The main steps highlighted on the simulation are explained below.

1. Control wrapper sender of IP core 00 sends an 'establishment packet' to control router of IP core 22. All the inner steps happen exactly as explained from steps 1 to 7 of the example of Section 6.6.1.
2. As soon as the control wrapper receiver of IP core 22 recognizes the 'establishment packet' after receiving the fourth flit of the packet, it asserts the *send_establishment_ack* signal high if the IP core is free to receive a communication (the *busy_IP* signal is not present on the figure).
3. Control wrapper sender of IP core 22 sends an 'acknowledgement packet' to control router of IP core 00. All the inner steps happen exactly as explained from steps 9 to 14 of the example of Section 6.6.1.
4. As soon as the control wrapper receiver of IP core 00 recognizes the 'acknowledgement packet' after receiving the fourth flit of the packet, it asserts the *establishment_done* signal high.
5. The IP core 00 sends the data communication to the IP core 22 through the data network as explained on steps 16 and 17 of the example of Section 6.6.1.
6. The control wrapper sender is informed by IP core 00 that all data was sent.
7. Control wrapper sender of IP core 10 sends an 'establishment packet' to control router 10 addressed to IP core 22.
8. Control router 10 starts retransmitting the 'establishment packet' to control router 11 three clock cycles after receiving it. The local input port is connected to the east output port on the data router 10.
9. Control router 11 retransmits the packet to control router 21. The west input port is connected to the east output port on the data router 11.
10. Control router 21 retransmits the packet to control router 22, but as soon as the control router recognizes the fourth flit of the packet as an 'establishment packet', it detects that the north output port of the data router is busy. Then the control router 21 transforms the 'establishment packet' into an 'establishment packet failed' by modifying the fourth flit from 00 to 10.
11. Control router 22 retransmits the 'establishment packet failed' to the control wrapper receiver 22.
12. The control wrapper receiver 22 receives the 'establishment packet failed' and activates the *send_router_busy* signal.
13. The control wrapper sender 22 transmits a 'not acknowledgement packet (router busy)' to control router 22.
14. Control router 22 retransmits the packet to control router 12.
15. Control router 12 retransmits the packet to control router 02.
16. Control router 02 retransmits the packet to control router 01.
17. Control router 01 retransmits the packet to control wrapper receiver 01.

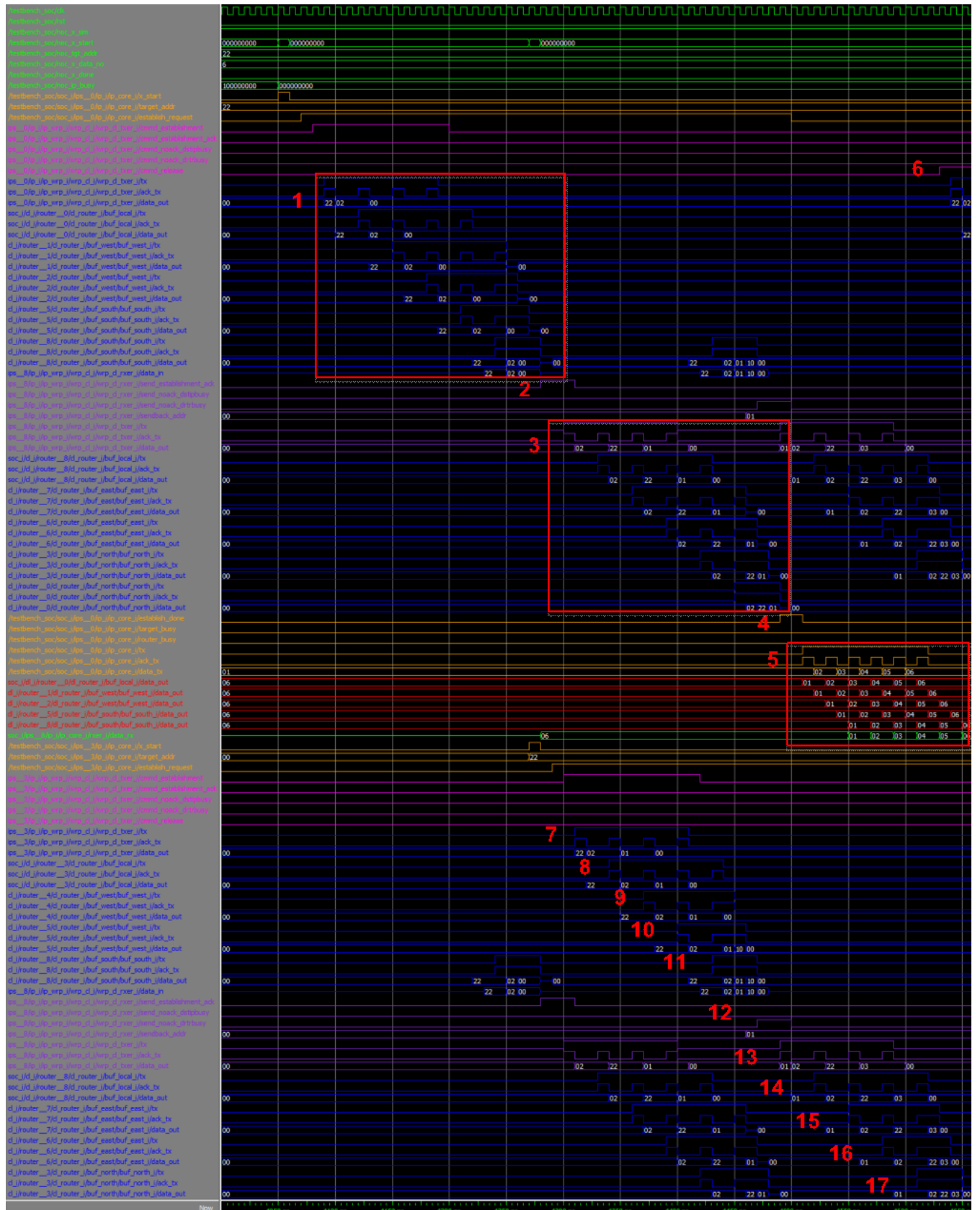


Fig. 6.19: Simulation of a situation where a data communication was not established because at least one intermediate data router required for the communication is busy.



18. As soon as the control wrapper receiver 01 recognizes the fourth flit of the packet as a 'not acknowledgement packet (router busy)', it informs the control wrapper sender by asserting high the signal *send_release* to disconnect the partially created data path between IP core 01 and IP core 22.
19. Control wrapper sender 01 transmits a 'release packet' to the control router 01 and targeted to the control wrapper 22.
20. Control router 01 retransmits the packet to control router 11. The local input port is disconnected of the east output port on the data router 01.
21. Control router 11 retransmits the packet to control router 21. The west input port is disconnected of the east output port on the data router 11.
22. Control router 21 retransmits the first 3 flits of the packet to control router 21. As soon as the fourth flit is recognized as a 'release packet' command and that the north output data port is not connected to the west input data port, the control router 21 transmits the fourth flit value as 14 instead of 04. This indicates that the packet is now a 'release packet blockage found' and that the next control routers should no longer disconnect any other segment of the communication.
23. Control router 22 retransmits the packet to control wrapper receiver 22.
24. Control wrapper receiver 22 receives the 'release packet blockage found' and do not perform any action.
25. The IP core 00 finishes to communicate with IP core 22 and all the inner steps happen exactly as explained from steps 18 to 25 of the example of Section 6.6.1.

6.7 Timing analysis of the SoC

In order to verify if the simulation presented on Section 6.6 reflects the reality, this Section presents the timing results of the real system executing on a ML507 evaluation platform from Xilinx, which contains a Virtex-5 FX70T FPGA. The system runs at 100MHz and the main signals of the system were probed with the ChipScope logic analyzer tool from Xilinx.

The same situation presented on Section 6.6.1, where IP core 00 communicates through the data network to IP core 22, is presented on Fig. 6.21. Please note that the clock is missing on the figure, please use the high state of an *ack_tx* signal as reference which is equal to the period of the clock. The main steps highlighted on the figure are explained below.

1. It takes 3 clock cycles for the first flit of a packet to arrive to a neighbor control router when there is no congestion.
2. It takes always 1 clock cycle for the first flit of a packet to arrive to a neighbor data router. It is always 1 clock cycle because the IP core source can only send data after the data path is reserved, so the throughput is guaranteed.
3. It has taken 44 clock cycles for IP core 00 to start transferring data to a target IP core 22, which are 4 hops apart from each other. Tab. 5.1 presents how every clock cycle is used.

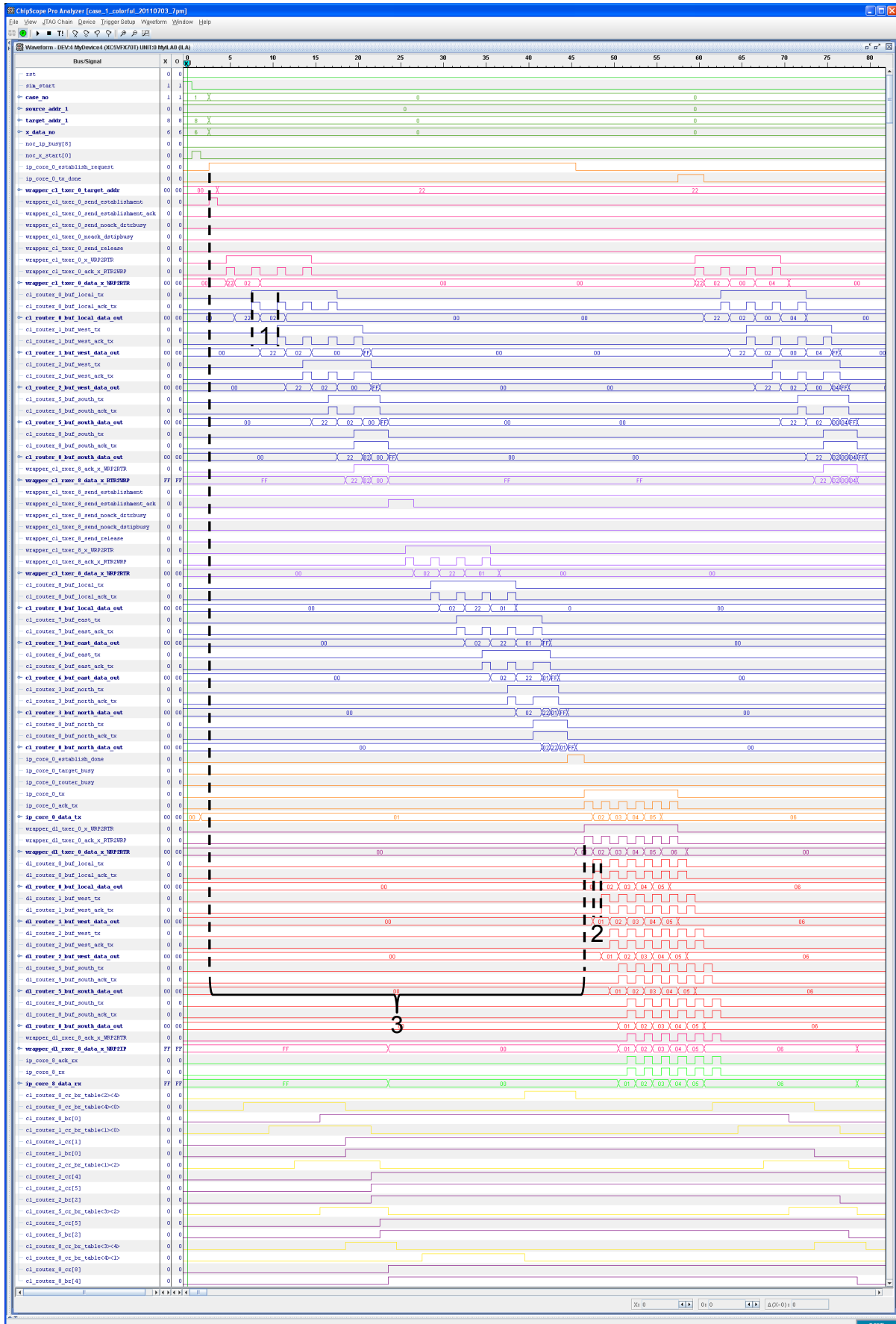


Fig. 6.21: ChipScope showing the protocol to perform a data communication between 2 IP cores.

Tab. 6.1: Amount of clock cycles used by different resources to establish a data communication between IP cores 00 and 22.

	latency	comment
route establishment	0	IP core 00 requests a data transfer
	2	Control wrapper sender 00 starts sending the establishment packet
	3	Control router 00 starts forwarding the packet
	3	Control router 10 starts forwarding the packet
	3	Control router 20 starts forwarding the packet
	3	Control router 21 starts forwarding the packet
	3	Control router 22 starts forwarding the packet
	4	Control wrapper receiver 22 recognizes the establishment packet
route acknowledgement	2	Control wrapper sender 22 starts sending the acknowledgement packet
	3	Control router 22 starts forwarding the packet
	3	Control router 12 starts forwarding the packet
	3	Control router 02 starts forwarding the packet
	3	Control router 01 starts forwarding the packet
	3	Control router 00 starts forwarding the packet
	4	Control wrapper receiver 00 recognizes the establishment packet
	2	IP core 00 starts sending data
	44	

From the presented timing information, it is possible to compare the improvements of this dual-layer NoC with the reconfigurable one presented on Chapter 5. Tab. 6.2 summarizes these improvements. The new control router takes half of the time to route a packet, which is due to the optimizations to reduce the buffer size from 8 flit buffers to 1 flit buffer and optimization on the routing algorithm. The time configure a data router is around 5,400,000 less than before, because now the control router directly configures the multiplexors of the data router and before an external computer was responsible to partially and dynamically reconfigure the data router. As a result, the time to establish a data path through the data network can be around 613,636 times less than before, when considering a data communication of two IP cores 4 hops apart from each other.

Tab. 6.2: Timing comparison between the reconfigurable dual-layer NoC presented on chapter 5 and the non reconfigurable dual-layer NoC presented on chapter 6.

	Reconfigurable Dual-layer NoC Chapter 5	Non Reconfigurable Dual-layer NoC Chapter 6
Time for a control router to route a packet	6	3
Time for configuring a data router	~5,400,000	1
Time for establishing a data path between two IP cores 4 hops apart from each other	~27,000,000	44
Time for forwarding a data flit	1	1

6.8 Area analysis

The floorplanning of the 3x3 dual-layer NoC presented on this Chapter is illustrated on

Fig. 6.22. The system was prototyped on a ML507 evaluation platform from Xilinx, which contains a Virtex-5 FX70T FPGA. Tab. 6.3 presents the resources used by each component of the system. The most important comparison that can be extract is that the data network is eight times smaller in terms of slices than the control network. This means that with a little bit more logic than what is used by the control network, each IP core could have its own dedicated network with guaranteed throughput to communicate with any other IP core. Of course that this is an extreme use of area, but it is an alternative to be considered before implementing NoCs with several virtual channels with huge buffers, because they do not really provided parallelism to transmit data, while the network presented here provides full parallelism and guaranteed throughput.

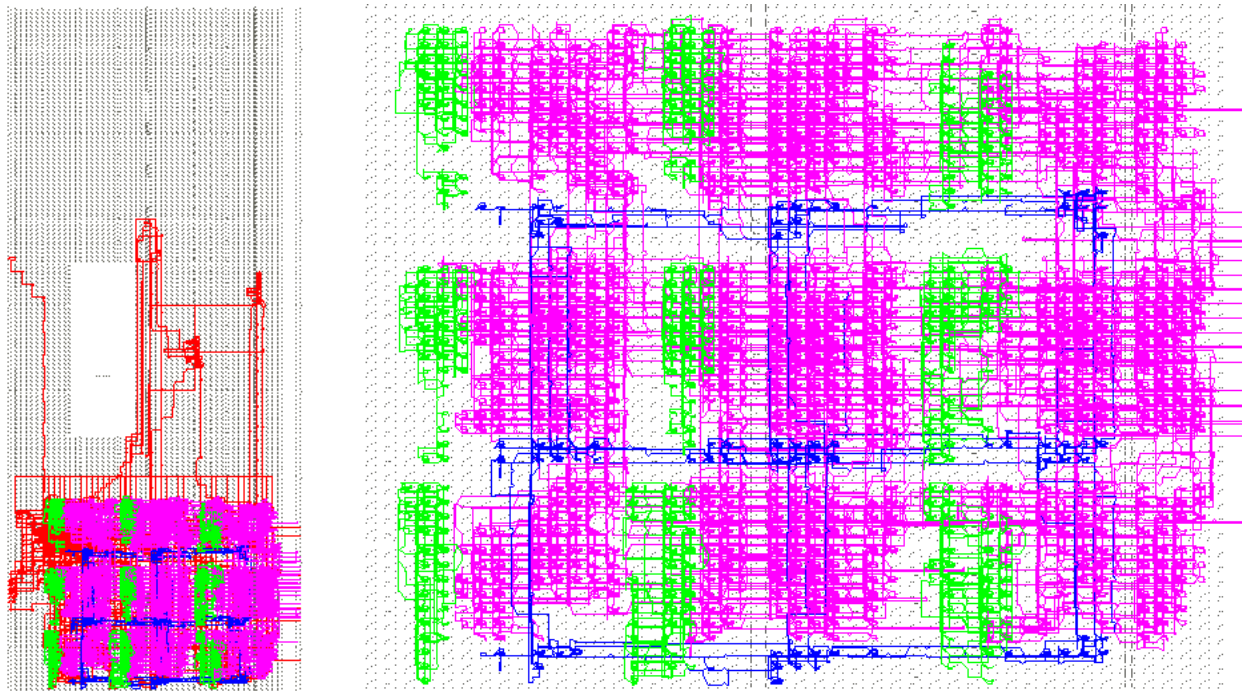


Fig. 6.22: Floorplanning of the 3x3 dual-layer NoC presented by the FPGA Editor tool from Xilinx. On the left, the complete overview of the project is presented. On the right, a zoom of the used area for the project is presented. The 9 control routers are depicted on pink, the 9 data routers are depicted on blue and the 9 IP cores including their wrappers and network interface are depicted on green.

Tab. 6.4 compares the use of area of the dual-layer NoC presented on Chapter 5 and the one presented on this Chapter. For this comparison to be fair, the non-reconfigurable dual-layer NoC presented on this Chapter was synthesized targeting a Virtex-4 FX12, exactly the same device used on Chapter 5. The table shows that the control routers of the non-reconfigurable network use 45% more flip-flops and 14% more LUTs, which makes sense since each control router requires extra logic to control and configure a data router. However, this comparison is unfair since the reconfigurable network requires a centralized configuration controller to configure the data routers, which is running on a computer and cannot have its area usage computed. On the other hand, the data router of the non-reconfigurable network uses 2% of the flips flops and 25% of the LUTs used by the reconfigurable network.

Tab. 6.3: Virtex-5 FX70T resource utilization of the dual-layer NoC.

	Flip-Flops	LUTs	Slices
Control Router 3 ports	120 (0.27%)	252 (0.56%)	109 (0.97%)
Control Router 4 ports	165 (0.37%)	396 (0.88%)	163 (1.46%)
Control Router 5 ports	193 (0.43%)	492 (1.1%)	185 (1.65%)
Data Router 3 ports	3 (0.01%)	18 (0.04%)	14 (0.13%)
Data Router 4 ports	4 (0.01%)	32 (0.07%)	19 (0.17%)
Data Router 5 ports	5 (0.01%)	46 (0.1%)	23 (0.2%)
Control Network 3x3	1406 (3.14%)	3233 (7.22%)	1346 (12.02%)
Data Network 3x3	33 (0.07%)	247 (0.55%)	161 (1.44%)
Dual-layer NoC	1439 (3.21%)	3480 (7.77%)	1507 (13.46%)
Dual-layer NoC + Test Cores	2532 (5.65%)	5033 (11.23%)	2082 (18.59%)
Amount available on FPGA	44800 (100%)	44800 (100%)	11200 (100%)

Tab. 6.4: Comparison of the area usage of the reconfigurable and non-reconfigurable dual-layer NoC.

	Reconfigurable network		Non-Reconfigurable network	
	Flip-Flops	LUTs	Flip-Flops	LUTs
Control Router 5 ports	132 (1.21%)	582 (5.31%)	192 (1.75%)	664 (6.07%)
Data Router 5 ports	224 (2.05%)	224 (2.05%)	5 (0.05%)	58 (0.53%)
Amount available on FPGA	10944 (100%)	10944 (100%)	10944 (100%)	10944 (100%)

6.9 Summary

This Chapter described a non-reconfigurable dual-layer NoC, where a packet switching NoC is used for control and a circuit switching NoC is used for data communication. The main novelty is that the control routers are used to configure the data routers on the data communication path between source and target of the communication. The advantage is that the crossbar, arbitration and routing algorithms can be removed from the circuit switching NoC, thus reusing these hardware blocks from the packet switching NoC. This idea could possibly be extended to control not only one circuit switching NoC, but several instantiations of the same or different types of NoCs. Another advantage is that after the path is set, the latency is one clock cycle per data router hop and the throughput is guaranteed.

The long waiting time to configure a data router on the reconfigurable dual-layer NoC presented on Chapter 5 was improved on this Chapter and is around 613,636 times faster. The main problem of the reconfigurable dual-layer NoC presented on Chapter 5 was that it relies on a centralized external computer to configure the data routers. Here, distributed control routers are used to configure the data routers. This allowed to configure the data routers on-chip, resulting in a configuration time of one clock cycle after its corresponding control router has received a request to configure it.

7 Conclusions and Future Works

The joint modeling of application, communication, monitoring and mapping allowed not only the convergence to an efficient MPSoC, but also the possibility to explore a wide range of design alternatives. Some key factors for providing valuable information regarding the MPSoC model are accuracy, flexibility and speed. Accuracy can be seen on the platform model by back annotating the timing behavior of the reference RTL NoC to the NoC model used by the MPSoC. Accuracy is achieved by modeling the application in detail (e.g. setting the size of the messages exchanged by tasks; setting the computation time of each task; setting the affinity, utilization and memory footprint of each task). Flexibility and speed are achieved by not setting all the details of the application, thus providing initial figures of the application's performance even when only rough estimations can be inserted into the model. Speed can also be achieved by using a more abstract communication platform like BOÇA. Accuracy can be achieved by using a more detailed communication platform like RENATO. Further accuracy can be achieved in communication platforms like RENATO or JOSELITO by fine-tuning parameters like buffer and flit sizes.

As a general rule it can be said that fast and abstract platform models should be used early on the design flow in order to perform rough evaluations and to rule out poorly performing platforms. Accurate models should be used later for fine-tuning platform parameters and choosing the best performance. All three platform models presented on this work were simulated with the same application but different mappings, and it was verified that the trends of latency are consistent across abstraction levels [112]. In addition, eight dynamic mapping algorithms and five task migration algorithms were presented, which can automate the process of verifying that each task of the application is running on an appropriate location. Eight types of monitoring systems can be attached to the MPSoC, providing not only a visual view of the execution of the MPSoC, but also a feedback to the dynamic mapping and task migration algorithms. The support to a heterogeneous MPSoC is achieved by an intelligent type system, which also receives a feedback from the monitoring system to assist the dynamic mapping or task migration algorithms.

Even though a lot of intelligence was added to the MPSoC model like the heterogeneous system and task mapping algorithms, the standard packet switching NoC limited the performance of the system by not providing any kind of priority or QoS for certain types of communication. For this reason, a dual-layer NoC composed by a packet switching

network and a circuit-switching network was implemented. The packet switching network should be mainly used for control purposes, but a limited number of small data packets can be accepted. The circuit-switching network provides by nature a guaranteed throughput service, which is a requirement of several embedded systems applications. It is important to highlight that this dual-layer NoC is not a simple connection of two types of NoCs, but the data network (implemented as a circuit switching NoC) was completely optimized for area and does not work without a control network (implemented as a packet switching NoC). For sure this has increased a little the area consumption of the control routers, but this has greatly reduced the area of the data routers while doubling the number of wires used for the communication purposes. And the main goal behind this new interconnection infrastructure is to be possible in a future work to easily parameterize the number of data networks of the system according to the requirements of the application, without increasing too much the area consumption.

Therefore, the primary future work is to model a multi-layer NoC based on the dual-layer NoC presented on this work. The major modification required would be a new communication protocol that can find one free data network to perform a communication between two IP cores. On such a system, maybe it would be useful for the control routers to accept not only distributed routing, but also source routing. Source routing can allow more advanced routes to be found than the standard XY routing algorithm, thus minimizing the number of data networks required.

This multi-layer NoC could be implemented on a 3D chip (also known as 3D integrated circuit), where the main advantage would be the decrease of interconnect length. This would in turn result in increased speed, better utilization of chip area and increased transistor packing densities, leading to better performance and power efficiency [72]. On such 3D chip, the multi-layer NoC could be implemented on several layers or only one layer of the 3D chip, while the other layers could integrate “dissimilar technologies” (e.g. memory, DSPs, MEMS) [73] as well as non-silicon [74].

So far, the type system created for the heterogeneous MPSoC model can only assign a type for an IP core at design time. A very interesting idea to improve the performance of the system is to adapt the type of an IP core according to its execution needs at runtime. One example is if only general purpose processors (GPP) are currently available in the platform, and a new task is triggered to be executed. This task can be implemented either by a GPP or a specialized hardware block. The specialized hardware block is able to run this task 10 times faster than the GPP. Then, one GPP could be made available by migrating its current tasks, and the IP core that implements this GPP could be dynamically changed to the type of the specialized hardware block. This is a promising idea that could be realized on a chip with the use of a partially and dynamically reconfigurable device (e.g. FPGA, eFPGA).

Another interesting future work would be to create a distributed MPSoC model, where one computer would be responsible for executing the multi-layer NoC and other computers connected to the same network would be responsible for the computing part. In this case, each IP core could be either an instruction set simulator pretending to be a

processor, or a special computer program pretending to be a specific hardware block connected to the MPSoC. Such a work could bring a speed-up for the simulation of applications composed by hundreds of IP cores or applications that need to be simulated for an extended time in order to provide any valuable result.

References

- [1] Intel Corporation, "Intel Pentium D (Smithfield) Processor," retrieved at: <http://ark.intel.com/ProductCollection.aspx?codeName=5788>, accessed on: Jun 2011.
- [2] C. Brooks, E.A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng, "Heterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to Ptolemy II)," UC Berkeley, 2003, 240 p.
- [3] E.A. Lee and S. Neuendorffer, "Actor-Oriented Models for Codesign: Balancing Re-Use and Performance," in *Formal Methods and Models for System Design*, Kluwer, 2004, pp. 33-56.
- [4] T. Bjerregaard and S. Mahadevan, "A Survey of Research and Practices of Network-on-Chip," *ACM Computing Surveys (CSUR)*, vol. 38(1), 2006, pp. 1-51.
- [5] R. Kourdy, S. Yazdanpanah, and M.R.N. Rad, "Using the NS-2 Network Simulator for Evaluating Multi Protocol Label Switching in Network-on-Chip," in *Conference on Computer Research and Development (CCRD)*, 2010, pp. 795-799.
- [6] A. Hegedus, G.M. Maggio, and L. Kocarev, "A NS-2 Simulator Utilizing Chaotic Maps for Network-on-Chip Traffic Analysis," in *International Symposium on Circuits and Systems (ISCAS)*, 2005, pp. 3375-3378.
- [7] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny, "QNoC: QoS Architecture and Design Process for Network on Chip," *Journal of Systems Architecture (JSA)*, vol. 50(2-3), Feb 2004, pp. 105-128.
- [8] J. Xu, W. Wolf, J. Henkel, and S. Chakradhar, "A Design Methodology for Application-Specific Networks-on-Chip," *ACM Transactions on Embedded Computing Systems*, vol. 5(2), May 2006, pp. 263-280.
- [9] G. Fen, W. Ning, and W. Qi, "Simulation and Performance Evaluation for Network on Chip Design Using OPNET," in *IEEE Region 10 Conference (TENCON)*, 2007, pp. 1-4.
- [10] H.Z. Zhao and Q. Wang, "Simulation and Evaluation for SS Network on Chip architecture using OPNET," in *International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*, 2006, pp. 2098-2101.
- [11] D. Bertozzi, A. Jalabert, S. Murali, R. Tamhankar, S. Stergiou, L. Benini, and G. De Micheli, "NoC Synthesis Flow for Customized Domain Specific Multiprocessor

- Systems-on-Chip," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 16(2), 2005, pp. 113–129.
- [12] T. Kogel, M. Doerper, A. Wieferink, R. Leupers, G. Ascheid, H. Meyr, and S. Goossens, "A Modular Simulation Framework for Architectural Exploration of On-Chip Interconnection Networks," in *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2003, pp. 7–12.
- [13] M. Coppola, S. Curaba, M.D. Grammatikakis, G. Maruccia, and F. Papariello, "OCCN: A Network on Chip Modeling and Simulation Framework," in *Design, Automation and Test in Europe (DATE)*, 2004, pp. 174–179.
- [14] F. Dumitrascu, I. Bacivarov, L. Pieralisi, M. Bonaciu, and A. Jerraya, "Flexible MPSoC Platform with Fast Interconnect Exploration for Optimal System Performance for a Specific Application," in *Design, Automation and Test in Europe (DATE)*, 2006, pp. 166–171.
- [15] S.G. Pestana, E. Rijpkema, K. Goossens, and O.P. Gangwal, "Cost-Performance Trade-Offs in Networks on Chip: A Simulation-Based Approach," in *Design, Automation and Test in Europe (DATE)*, 2004, pp. 1–6.
- [16] P.P. Pande, C. Grecu, M. Jones, A. Ivanov, and R. Saleh, "Performance Evaluation and Design Trade-Offs for Network-on-Chip Interconnect Architectures," *IEEE Transactions on Computers*, vol. 54(8), Aug 2005, pp. 1025–1040.
- [17] L. Benini and G. De Micheli, "Networks on Chips: A New SoC Paradigm," *IEEE Computer*, vol. 35(1), 2002, pp. 70–78.
- [18] L.S. Indrusiak, R.B. Prudencio, and M. Glesner, "Modeling and Prototyping of Communication Systems Using Java: A Case Study," in *International Symposium on Rapid System Prototyping (RSP)*, 2005, pp. 225–231.
- [19] L.S. Indrusiak, A. Thuy, and M. Glesner, "Executable System-Level Specification Models Containing UML-Based Behavioral Patterns," in *Design, Automation and Test in Europe (DATE)*, 2007, pp. 301–306.
- [20] A. Knapp and J. Wuttke, "Model Checking of UML 2.0 Interactions," in *Models in Software Engineering, Workshops and Symposia (MODELS)*, 2006, pp. 42–51.
- [21] G. Martin, "Overview of the MPSoC Design Challenge," in *Design Automation Conference (DAC)*, 2006, pp. 274–279.
- [22] T. Marescaux, A. Rangevall, V. Nollet, A.T. Bartic, and H. Corporaal, "Distributed Congestion Control for Packet Switched Networks on Chip," in *Parallel Computing Conference (ParCo)*, 2005, pp. 761–768.
- [23] A.W. Yin, L. Guang, P. Liljeberg, P. Rantala, E. Nigussie, J. Isoaho, and H. Tenhunen, "Hierarchical Agent Architecture for Scalable NoC Design with Online Monitoring Services," in *International Workshop on Network on Chip Architectures (NoCArc)*, 2008, pp. 58–63.
- [24] C. Ciordas, T. Basten, A. Radulescu, K. Goossens, and J. Meerbergen, "An Event-Based Network-on-Chip Monitoring Service," in *International High-Level Design Validation and Test Workshop*, 2004, pp. 149–154.

- [25] C. Ciordas, A. Hansson, K. Goossens, and T. Basten, "A Monitoring-Aware Network-on-Chip Design Flow," *Journal of Systems Architecture (JSA)*, vol. 54/2008, pp. 397-410.
- [26] J.C.S.A. Palma, L.S. Indrusiak, F.G. Moraes, A.G. Ortiz, M. Glesner, and R. Reis, "Adaptive Coding in Networks-on-Chip Transition Activity Reduction versus Power Overhead of the Codec Circuitry," in *International Workshop on Power And Timing Modeling, Optimization and Simulation (PATMOS)*, 2006, pp. 603-613.
- [27] L. Tedesco, A.V.D. Mello, L. Giacomet, N.L.V. Calazans, and F.G. Moraes, "Application Driven Traffic Modeling for NoCs," in *Symposium on Integrated Circuits and Systems Design (SBCCI)*, 2006, pp. 62-67.
- [28] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri, "MPARM: Exploring the Multi-Processor SoC Design Space with SystemC," *The Journal of VLSI Signal Processing*, vol. 41(2), 2005, pp. 169-182.
- [29] S. Boukhechem and E.-bay Bourennane, "SystemC Transaction-Level Modeling of an MPSoC Platform Based on an Open Source ISS by Using Interprocess Communication," *International Journal of Reconfigurable Computing*, vol. 2008/2008, pp. 1-10.
- [30] J. Ceng, W. Sheng, J. Castrillon, A. Stulova, R. Leupers, G. Ascheid, and H. Meyr, "A High-Level Virtual Platform for Early MPSoC Software Development," in *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2009, pp. 11-20.
- [31] N. Pouillon, A. Becoulet, A.V.D. Mello, F. Pecheux, and A. Greiner, "A Generic Instruction Set Simulator API for Timed and Untimed Simulation and Debug of MP2-SoCs," in *International Symposium on Rapid System Prototyping (RSP)*, 2009, pp. 116-122.
- [32] SoCLib, "SoCLib," retrieved at: <http://www.soclib.fr>, accessed on: May 2011.
- [33] Opencores, "Opencores," retrieved at: <http://www.opencores.org>, accessed on: May 2011.
- [34] K. Vollmar and P. Sanderson, "A MIPS Assembly Language Simulator Designed for Education," *Journal of Computing Sciences in Colleges*, vol. 21(1), 2005, pp. 95-101.
- [35] J. Eker, J.W. Janneck, E.A. Lee, J. Ludvig, S. Neuendorffer, and S. Sachs, "Taming Heterogeneity - The Ptolemy Approach," *Proceedings of the IEEE*, vol. 91(1), Jan 2003, pp. 127-144.
- [36] E.A. Carara, R.P. de Oliveira, N.L.V. Calazans, and F.G. Moraes, "HeMPS - A Framework for NoC-Based MPSoC Generation," in *International Symposium on Circuits and Systems (ISCAS)*, 2009, pp. 1345-1348.
- [37] Peter Sanderson and Kenneth Vollmar, "MARS MIPS Simulator," retrieved at: <http://courses.missouristate.edu/KenVollmar/MARS/>, accessed on: May 2011.
- [38] OMG, "OMG Unified Modeling Language Specification (v1.4 draft)," OMG, 2001, 576 p.

- [39] L.S. Indrusiak and M. Glesner, "SoC Specification using UML and Actor-Oriented Modeling," in *Biennial Baltic Electronics Conference (BEC)*, 2006, pp. 1-6.
- [40] L.S. Indrusiak and M. Glesner, "Specification of Alternative Execution Semantics of UML Sequence Diagrams within Actor-Oriented Models," in *Symposium on Integrated Circuits and Systems Design (SBCCI)*, 2007, pp. 330-335.
- [41] Luciano Copello Ost, "Abstract Models of NoC-Based MPSoCs for Design Space Exploration," PhD Thesis, Catholic University of Rio Grande do Sul, 2010, 98 p.
- [42] H. Blume, H. Hubert, H.T. Feldkamper, and T.G. Noll, "Model-Based Exploration of the Design Space for Heterogeneous Systems on Chip," in *International Conference on Application Specific Array Processors (ASAP)*, 2002, pp. 29-40.
- [43] E.L. de S. Carvalho, N.L.V. Calazans, and F.G. Moraes, "Dynamic Task Mapping for MPSoCs," *IEEE Design & Test of Computers*, vol. 27(5), Sep 2010, pp. 26-35.
- [44] M.A. Faruque, R. Krist, and J. Henkel, "ADAM: Run-Time Agent-Based Distributed Application Mapping for On-Chip Communication," in *Design Automation Conference (DAC)*, 2008, pp. 760-765.
- [45] A. Weichslgartner, S. Wildermann, and J. Teich, "Dynamic Decentralized Mapping of Tree-Structured Applications on NoC Architectures," in *International Symposium on Networks-on-Chip (NOCS)*, 2011, pp. 201-208.
- [46] S. Wildermann, T. Ziermann, and J. Teich, "Run time Mapping of Adaptive Applications onto Homogeneous NoC-based Reconfigurable Architectures," in *International Conference on Field-Programmable Technology (FPT)*, 2009, pp. 514-517.
- [47] A. Molnos, J.A. Ambrose, A. Nelson, R. Stefan, S. Cotofana, and K. Goossens, "A Composable, Energy-Managed, Real-Time MPSoC Platform," in *International Conference on Optimization of Electrical and Electronic Equipment (OPTIM)*, 2010, pp. 870-876.
- [48] C.-L. Chou and R. Marculescu, "Run-Time Task Allocation Considering User Behavior in Embedded Multiprocessor Networks-on-Chip," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 29(1), Jan 2010, pp. 78-91.
- [49] A.K. Singh, T. Srikanthan, A. Kumar, and W. Jigang, "Communication-Aware Heuristics for Run-Time Task Mapping on NoC-based MPSoC Platforms," *Journal of Systems Architecture*, vol. 56(7), Jul 2010, pp. 242-255.
- [50] P.K.F. Hölzenspies, J.L. Hurink, J. Kuper, and G.J.M. Smit, "Run-time Spatial Mapping of Streaming Applications to a Heterogeneous Multi-Processor System-on-Chip (MPSoC)," in *Design, Automation and Test in Europe (DATE)*, 2008, pp. 212-217.
- [51] L.T. Smit, J.L. Hurink, and G.J.M. Smit, "Run-time Mapping of Applications to a Heterogeneous SoC," in *International Symposium on System-on-Chip (ISSOC)*, 2005, pp. 78-81.
- [52] F. Ferrandi, P.L. Lanzi, C. Pilato, D. Sciuto, and A. Tumeo, "Ant Colony Heuristic for Mapping and Scheduling Tasks and Communications on Heterogeneous

- Embedded Systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29(6), Jun 2010, pp. 911-924.
- [53] L. Huang, R. Ye, and Q. Xu, "Customer-Aware Task Allocation and Scheduling for Multi-Mode MPSoCs," in *Design Automation Conference (DAC)*, 2011, pp. 387-392.
- [54] T. Kempf, K. Karuri, S. Wallentowitz, G. Ascheid, R. Leupers, and H. Meyr, "A SW Performance Estimation Framework for Early System-Level-Design Using Fine-Grained Instrumentation," in *Design, Automation and Test in Europe (DATE)*, 2006, pp. 468-473.
- [55] B. Ristau, T. Limberg, and G. Fettweis, "A Mapping Framework for Guided Design Space Exploration of Heterogeneous MP-SoCs," in *Design, Automation and Test in Europe (DATE)*, 2008, pp. 780-783.
- [56] T. Lei and S. Kumar, "A Two-Step Genetic Algorithm for Mapping Task Graphs to a Network on Chip Architecture," in *Conference on Digital System Design (DSD)*, 2003, pp. 180-187.
- [57] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio, "A SoC Design Methodology Involving a UML 2.0 Profile for SystemC," in *Design, Automation and Test in Europe (DATE)*, 2005, pp. 704-709.
- [58] T. Arpinen, P. Kukkala, E. Salminen, M. Hannikainen, and T.D. Hamalainen, "Configurable Multiprocessor Platform with RTOS for Distributed Execution of UML 2.0 Designed Applications," in *Design, Automation and Test in Europe (DATE)*, 2006, pp. 1324-1329.
- [59] E.L.D.S. Carvalho, N.L.V. Calazans, and F.G. Moraes, "Heuristics for Dynamic Task Mapping in NoC-based Heterogeneous MPSoCs," in *International Symposium on Rapid System Prototyping (RSP)*, 2007, pp. 34-40.
- [60] D. Choffnes, M. Astley, and M.J. Ward, "Migration Policies for Multi-Core Fair-Share Scheduling," *ACM SIGOPS Operating Systems Review*, vol. 42(1), Jan 2008, pp. 92-93.
- [61] A. Chandra, M. Adler, P. Goyal, and P. Shenoy, "Surplus Fair Scheduling: A Proportional-Share CPU Scheduling Algorithm for Symmetric Multiprocessors," in *Conference on Symposium on Operating System Design & Implementation (OSDI)*, 2000, pp. 45-58.
- [62] Xilinx, "Early Access Partial Reconfiguration (User Guide 208 v1.2)," Technical Report, Xilinx Inc, 2008, 64 p.
- [63] B. Jackson, "Partial Reconfiguration Design with PlanAhead," Technical Report, Xilinx Inc, 2008, 26 p.
- [64] H. Mrabet, Z. Marrakchi, H. Mehrez, and A. Tissot, "Implementation of Scalable Embedded FPGA for SOC," in *International Conference on Design and Test of Integrated Systems in Nanoscale Technology (DTIS)*, 2006, pp. 74-77.
- [65] P. Athanas and H.F. Silverman, "Processor Reconfiguration Through Instruction-Set Metamorphosis," *IEEE Computer*, vol. 13(3), 1993, pp. 11-18.

- [66] S.C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R.R. Taylor, and R. Laufer, "PipeRench: A Coprocessor for Streaming Multimedia Acceleration," in *International Symposium on Computer Architecture (ISCA)*, 1999, pp. 28–39.
- [67] Xilinx, "Xilinx UG080 ML401/ML402/ML403 Evaluation Platform User Guide," Technical Report, Xilinx Inc, 2006, 32 p.
- [68] GAPH, "Hardware Design Support Group," retrieved at: <http://www.inf.pucrs.br/~gaph>, accessed on: Oct 2011.
- [69] C. Claus, B. Zhang, W. Stechele, L. Braun, M. Hübner, and J. Becker, "A Multi-Platform Controller Allowing for Maximum Dynamic Partial Reconfiguration Throughput," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2008, pp. 535–538.
- [70] T. Becker, W. Luk, and P.Y.K. Cheung, "Enhancing Relocatability of Partial Bitstreams for Run-Time Reconfiguration," in *Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2007, pp. 35–44.
- [71] Mentor Graphics, "ModelSim," retrieved at: <http://model.com>, accessed on: Jun 2011.
- [72] K. Banerjee, S.J. Souri, P. Kapur, and K.C. Saraswat, "3-D ICs: A Novel Chip Design for Improving Deep-Submicrometer Interconnect Performance and Systems-on-Chip Integration," *Proceedings of the IEEE*, vol. 89(5), May 2001, pp. 602–633.
- [73] I. Anagnostopoulos, A. Bartzas, I. Vourkas, and D. Soudris, "Node Resource Management for DSP Applications on 3D Network-on-Chip Architecture," in *International Conference on Digital Signal Processing*, 2009, pp. 1–6.
- [74] V.F. Pavlidis and E.G. Friedman, "3-D Topologies for Networks-on-Chip," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15(10), Oct 2007, pp. 1081–1090.

Related List of Own Publications

- [101] L.C. Ost, F.G. Moraes, L.H. Möller, L.S. Indrusiak, M. Glesner, S. Määttä, and J. Nurmi, "A Simplified Executable Model to Evaluate Latency and Throughput of Networks-on-Chip," in *Symposium on Integrated Circuits and Systems Design (SBCCI)*, 2008, pp. 170-175.
- [102] L.S. Indrusiak, L.C. Ost, L.H. Möller, F.G. Moraes, and M. Glesner, "Applying UML Interactions and Actor-oriented Simulation to the Design Space Exploration of Network-on-Chip Interconnects," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2008, pp. 491-494.
- [103] S. Varyani, L.S. Indrusiak, T. Lui, L. Ost, L.H. Möller, and M. Glesner, "Experimental Review of Task Mapping Algorithms for NoC-Based Multiprocessor Systems-on-Chip," in *International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC)*, 2008, p. 4.
- [104] S. Määttä, L.S. Indrusiak, L.C. Ost, L.H. Möller, J. Nurmi, M. Glesner, and F.G. Moraes, "Validation of Executable Application Models Mapped onto Network-on-Chip Platforms," in *International Symposium on Industrial Embedded Systems (SIES)*, 2008, pp. 118-125.
- [105] S. Määttä, L.S. Indrusiak, L.C. Ost, L.H. Möller, M. Glesner, F.G. Moraes, and J. Nurmi, "Characterising Embedded Applications Using a UML Profile," in *International Symposium on System-on-Chip (ISSOC)*, 2009, pp. 172-175.
- [106] L.H. Möller, L.S. Indrusiak, and M. Glesner, "NoCScope: A Graphical Interface to Improve Networks-on-Chip Monitoring and Design Space Exploration," in *International Design and Test Workshop (IDT)*, 2009, pp. 1-6.
- [107] H. Yu, P.H.W. Leong, H. Hinkelmann, L.H. Möller, M. Glesner, and P. Zipf, "Towards a Unique FPGA-Based Identification Circuit Using Process Variations," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2009, pp. 397-402.
- [108] L.H. Möller, A. Rodrigues, F.G. Moraes, L.S. Indrusiak, and M. Glesner, "Instruction Set Simulator for MPSoCs based on NoCs and MIPS Processors," in *International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC)*, 2010, pp. 7-12.
- [109] L.H. Möller, P. Fischer, F.G. Moraes, L.S. Indrusiak, and M. Glesner, "Improving QoS of Multi-Layer Networks-on-Chip with Partial and Dynamic Reconfiguration of Routers," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2010, p. 229-233.

- [110] L.H. Möller, H.V. Jesus, F.G. Moraes, L.S. Indrusiak, T. Hollstein, and M. Glesner, "Graphical Interface for Debugging RTL Networks-on-Chip," in *Biennial Baltic Electronics Conference (BEC)*, 2010, p. 181–184.
- [111] L.S. Indrusiak, L.C. Ost, F.G. Moraes, S. Määttä, J. Nurmi, L.H. Möller, and M. Glesner, "Evaluating the Impact of Communication Latency on Applications Running over On-Chip Multiprocessing Platforms: A Layered Approach," in *International Conference on Industrial Informatics (INDIN)*, 2010, p. 148–153.
- [112] S. Määttä, L.H. Möller, L.S. Indrusiak, L.C. Ost, F.G. Moraes, J. Nurmi, and M. Glesner, "Joint Validation of Application Models and Multi-Abstraction Network-on-Chip Platforms," *International Journal of Embedded and Real-Time Communication Systems (IJERTCS)*, vol. 1(1), 2010, pp. 86–101.
- [113] S. Määttä, L.S. Indrusiak, L.C. Ost, L.H. Möller, M. Glesner, F.G. Moraes, and J. Nurmi, "A Case Study of Hierarchically Heterogeneous Application Modelling Using UML and Ptolemy II," in *International Symposium on System-on-Chip (ISSOC)*, 2010, p. 68–71.
- [114] L.C. Ost, M. Mandelli, G.M. Almeida, L.S. Indrusiak, L.H. Möller, M. Glesner, G. Sassatelli, M. Robert, and F.G. Moraes, "Exploring Dynamic Mapping Impact on NoC-based MPSoCs Performance Using a Model-based Framework," in *Symposium on Integrated Circuits and Systems Design (SBCCI)*, 2011, p. 6.
- [115] L.H. Möller, L.S. Indrusiak, L.C. Ost, G. Sassatelli, F.G. Moraes, M. Glesner, "Comparative Analysis of Dynamic Task Mapping Heuristics on the Reduction of Response Time in Heterogeneous NoC-based MPSoCs," submitted to *Design, Automation and Test in Europe (DATE)*, 2012.

Unrelated List of Own Publications

- [201] D. Mesquita, F.G. Moraes, L.H. Möller, and N.L.V. Calazans, "Reconfiguração Parcial e Remota de Dispositivos FPGA da Família Virtex," in *Seminário de Computação Reconfigurável*, 2001.
- [202] D. Mesquita, F.G. Moraes, J.C.S.A. Palma, and L.H. Möller, "Reconfiguração Parcial e Remota de Cores FPGAs," in *Workshop IBERCHIP*, 2001.
- [203] J.C.S.A. Palma, A.V.D. Mello, L.H. Möller, F.G. Moraes, and N.L.V. Calazans, "Core Communication Interface for FPGAs," in *Symposium on Integrated Circuits and Systems Design (SBCCI)*, 2002, pp. 183-188.
- [204] L.H. Möller, D. Mesquita, and F.G. Moraes, "Tool-Set for Remote and Partial Reconfiguration," in *South Symposium on Microelectronics (SIM)*, 2002, pp. 127-130.
- [205] F.G. Moraes, D. Mesquita, J.C.S.A. Palma, L.H. Möller, and N.L.V. Calazans, "Development of a Tool-Set for Remote and Partial Reconfiguration of FPGAs," in *Design, Automation and Test in Europe (DATE)*, 2003, pp. 1122-1123.
- [206] F.G. Moraes, A.V.D. Mello, L.H. Möller, L.C. Ost, and N.L.V. Calazans, "Networks on Chip: Architecture and Prototyping," in *South Symposium on Microelectronics (SIM)*, 2003, p. 5.
- [207] D. Mesquita, F.G. Moraes, J.C.S.A. Palma, L.H. Möller, and N.L.V. Calazans, "Remote and Partial Reconfiguration of FPGAs: Tools and Trends," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2003, p. 8.
- [208] F.G. Moraes, A.V.D. Mello, L.H. Möller, L.C. Ost, and N.L.V. Calazans, "A Low Area Overhead Packet-switched Network on Chip: Architecture and Prototyping," in *International Conference on Very Large Scale Integration (VLSI-SOC)*, 2003, pp. 318-323.
- [209] A.V.D. Mello, L.H. Möller, and F.G. Moraes, "Desenvolvimento de um Sistema Multiprocessado para Dispositivos FPGAs," in *Workshop IBERCHIP*, 2003.
- [210] J.C.S.A. Palma, A.V.D. Mello, L.H. Möller, F.G. Moraes, and N.L.V. Calazans, "Core Communication Interface for FPGAs," *Journal of Integrated Circuits and Systems*, vol. 1(1), 2004, pp. 44-51.
- [211] E.L.D.S. Carvalho, E.W. Brião, L.H. Möller, F.B. Möller, F.G. Moraes, and N.L.V. Calazans, "Controlling Configurations on Dynamic Reconfigurable Systems," in *South Symposium on Microelectronics (SIM)*, 2004, pp. 114-119.
- [212] L.H. Möller, N.L.V. Calazans, F.G. Moraes, E.W. Brião, E.L.D.S. Carvalho, and D. Camozzato, "FiPre: An Implementation Model to Enable Self-Reconfigurable

Applications,” in *International Conference on Field Programmable Logic and Applications (FPL)*, 2004, p. 1042–1046.

[213] E.W. Brião, E.L.D.S. Carvalho, L.H. Möller, D. Camozzato, N.L.V. Calazans, and F.G. Moraes, “A Generic Model of Embedded System to Enable Dynamic Self-Reconfigurable Applications,” in *South Symposium on Microelectronics (SIM)*, 2004, pp. 98–104.

[214] L.H. Möller, A.V.D. Mello, and E.A. Carara, “MultiNoC: A Multiprocessing System Enabled by a Network on Chip,” Technical Report, Catholic University of Rio Grande do Sul, 2004, 15 p.

[215] F.G. Moraes, N.L.V. Calazans, A.V.D. Mello, L.H. Möller, and L.C. Ost, “HERMES: An Infrastructure for Low Area Overhead Packet-Switching Networks on Chip,” *Integration, the VLSI Journal*, vol. 38(1), Oct 2004, pp. 69–93.

[216] A.V.D. Mello, L.H. Möller, N.L.V. Calazans, and F.G. Moraes, “MultiNoC: A Multiprocessing System Enabled by a Network on Chip,” in *Design, Automation and Test in Europe (DATE)*, 2005, pp. 234–239.

[217] F.G. Moraes, N.L.V. Calazans, L.H. Möller, E.W. Brião, and E.L.D.S. Carvalho, “Dynamic and Partial Reconfiguration in FPGA SoCs: Requirements and Tools,” in *New Algorithms, Architectures and Applications for Reconfigurable Computing*, 2005, pp. 1–12.

[218] L.H. Möller, F.G. Moraes, and N.L.V. Calazans, “Processadores Reconfiguráveis Estado da Arte,” in *Workshop IBERCHIP*, 2005, pp. 110–113.

[219] L.H. Möller, R. Soares, E.L.D.S. Carvalho, I. Grehs, N.L.V. Calazans, and F.G. Moraes, “Infrastructure for Dynamic Reconfigurable Systems: Choices and Trade-offs,” in *Symposium on Integrated Circuits and Systems Design (SBCCI)*, 2006, pp. 44–49.

[220] L.H. Möller, I. Grehs, N.L.V. Calazans, and F.G. Moraes, “Reconfigurable Systems Enabled by a Network-on-Chip,” in *International Conference on Field Programmable Logic and Applications (FPL)*, 2006, pp. 1042–1046.

[221] L.H. Möller, I. Grehs, E.L.D.S. Carvalho, R. Soares, N.L.V. Calazans, and F.G. Moraes, “A NoC-Based Infrastructure to Enable Dynamic Self Reconfigurable Systems,” in *International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC)*, 2007, p. 23–30.

Supervised Theses

- [301] Peter Fischer, "Design and Evaluation of a Reconfigurable Network-on-Chip Router Using the FPGA Early Access Partial Reconfiguration Flow," Master Thesis, Darmstadt University of Technology, 2009, 84 p.
- [302] André Rodrigues, "Instruction Set Simulator for MPSoCs based on NoCs and MIPS," Studienarbeit, Darmstadt University of Technology, 2010, 46 p.
- [303] Hélio Vieira Jesus, "Graphical Interface for Debugging the Hermes NoC," Studienarbeit, Darmstadt University of Technology, 2010, 39 p.
- [304] Fouad Korkmaz, "Algorithms for Task Migration on Actor-Oriented Multiprocessor Systems-on-Chip," Master Thesis, Darmstadt University of Technology, 2010, 53 p.
- [305] He Zhou, "Design and Evaluation of a Multi-Layer Packet and Circuit Switched Network-on-Chip," Master Thesis, Darmstadt University of Technology, 2010, 67 p.
- [306] Wei Lin, "Ring Oscillators as Thermal Sensors for FPGAs," Studienarbeit, Darmstadt University of Technology, 2011, 61 p.
- [307] Muhammad Faraz Ishrat, "Development of a Heterogeneous Multiprocessor Systems-on-chip on an Actor-Oriented Platform," Master Thesis, Darmstadt University of Technology, 2011, 54 p.
- [308] Mohamad-Elmoubarak Bai, "Applications for Actor-Oriented Multiprocessors System-on-Chip," Bachelor Thesis, Darmstadt University of Technology, 2011, 117 p.
- [309] Thiago Erik Petersen, "Integration of a Graphical Interface Debugger to the Atlas Framework," Studienarbeit, Darmstadt University of Technology, 2011, 35 p.
- [310] Yi Xie, "Implementation and Evaluation of Dynamic Mapping Algorithms for Actor-Oriented Multiprocessor Systems-on-Chip," Diploma Thesis, Darmstadt University of Technology, 2011, 58 p.

Curriculum Vitae

Personal Data:

Name: Leandro Heleno Möller
Data of birth: February 7th, 1981
Place of birth: Porto Alegre, Brazil

Academic Formation:

1988 – 1995	Elementary school at “Colégio Bom Conselho” in Porto Alegre
1996 – 1998	High school at “Colégio João Paulo I” in Porto Alegre
1999 – 2003	Bachelor in Computer Science at “Pontifícia Universidade Católica do Rio Grande do Sul” in Porto Alegre
2004 – 2005	Master in Computer Science at “Pontifícia Universidade Católica do Rio Grande do Sul” in Porto Alegre
2007 – 2011	Ph.D. student, teaching and research assistant at the Institute of Microelectronic Systems, Darmstadt University of Technology, Darmstadt, Germany

Erklärung laut §9 PromO

Ich versichere hiermit, dass ich die vorliegende Dissertation allein und nur unter Verwendung der angegebenen Literatur verfasst habe. Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

24.10.2011, Leandro Heleno Möller